Theorem Proving: Not an Esoteric Diversion, but the Unifying Framework for Industrial Verification*

To appear in proceedings of IEEE International Conference on Computer Design (ICCD) '95
Austin, Texas October 1995

D. A. Cyrluk
SRI International and
Computer Science Department
Stanford University
Stanford, CA 94305

M. K. Srivas

Computer Science Laboratory
SRI International
Menlo Park, CA 94025

Abstract

The effectiveness of hardware verification techniques has increased markedly in the past decade. As hardware verification techniques become increasingly powerful the idea of transitioning verification technology to industry can be taken seriously. Nevertheless, powerful decision procedures that can completely automate the verification of certain types of hardware, whether they are BDD based model-checkers [10] or automatic microprocessor verification tools [4], cannot be adequate on their own for industrial hardware verification. However, a high-level, general-purpose theorem prover with specific capabilities can provide an overall framework in which these tools can be embedded and in which they can then be effectively used for industrial hardware verification.

1 Introduction

The effectiveness of hardware verification techniques has increased markedly in the past decade. These techniques span a spectrum from automatic techniques to interactive theorem proving techniques. At one end of the spectrum automatic methods such as model-checking have improved considerably: model-checkers have been able to verify circuits with 10^{20} states [3] and have verified the correctness of a PDP-11 sized processor [2], special purpose tools have been able to automate the verification of complete benchmark microprocessors [4, 6, 7], language containment methods [10] have also been effectively applied to significant examples. At the other end of the spectrum, the Boyer-Moore theorem prover and the HOL theorem prover have been successfully used to verify simple microprocessor [11, 12]. Most recently, PVS [16], a higher-order, general-purpose theorem prover being developed at SRI, has verified a large part of the AAMP5—a half million transistor, commercial pipelined microcoded microprocessor developed at Rockwell-Collins [14].

As hardware verification techniques become increasingly powerful we are approaching a point where the idea of transitioning the technology to industry can be taken seriously. An important question that this possibility raises is what type of verification system will industry find useful and effective. On the one hand, will it look like an essentially automatic verification tool such as model-checking that has been beefed up to the point that it can operate on industrial problems? Or will it be an interactive system that brings together various tools and methods in an integrated fashion? What role will theorem proving play in such a tool?

Our view is that a high-level general-purpose theorem prover with certain capabilities (to be discussed) can function as a tool to effectively transition hardware verification to industry.

We have argued [7] that to be effective at hardware verification theorem provers must provide powerful and efficient primitive inference procedures that can be combined by means of user-defined general-purpose, high-level proof strategies. Our discussion [7] focused on the importance of powerful and efficient primitive inference procedures as these are generally lacking in many general-purpose interactive verification systems.

We argue in this paper, that the dual is equally important: that powerful decision procedures that can completely automate the verification of certain types of hardware such as a BDD based model checker [3] or automatic microprocessor verification tool [4] cannot be adequate on their own for industrial hardware verification.

Consider the following typical verification cycle employed by users of a model-checker or similar automatic tool. The first step might be to completely specify the circuit to be verified in the model-checking specification language, then specify the correctness property, and then invoke the model-checker. Due to the size and complexity of the circuit, the model-checker is not likely to return at this point. The user can now take several different paths. He can simplify the circuit by abstracting part of it away. He can reduce the state space by reducing the width of the data-paths. He can use abstraction techniques [5] to generate a smaller, but logically equivalent state

*This work was supported by NASA Langley Research Center under contract NAS1-20334 and NAS1-19704.

space. He can break up the problem into smaller subproblems, each of which he would attack separately. He might want to apply different tools to the different subproblems. Some of these tasks are designed to provide greater confidence to the user that his circuit is correct. Some are useful as a step in the verification of the whole circuit. However, the user must keep track of what he is doing on pencil and paper, or more likely in his head.

An interactive theorem prover can provide a framework in which to perform these different functions, with possibly different tools, starting with one specification. The theorem prover provides a principled way to manage the interaction of different verification tasks and, hopefully, in the end to combine them into a complete proof. Even when the verification is left incomplete a theorem prover provides information on which part of the verification task still needs to be done.

Theorem provers provide additional support for the verification cycle. Certain language features found in some theorem provers support large scale, industrial verification efforts. Just as programming using modules and object oriented constructs aid the construction of large software, so does using parameterized theories and a rich expressive type system aid in the specification, maintainance, and reuse of large specifications.

In the rest of the paper we expand on these ideas. We argue for three main points. First, theorem provers need not be an esoteric research tool, but can be effectively used by real hardware engineers. Second, there are aspects of hardware that makes verification a practical possibility. Third, aspects of industrial hardware verification make the use of an interactive general-purpose theorem prover a natural, effective, and necessary tool in which to tie together other automatic verification tools so as to perform industrial hardware verification.

In the next section we first address common concerns about using a theorem prover for real-world industrial tasks. In section 3 we give an overview of some features of hardware that might make the verification problem practically doable. Section 4 presents what we consider important aspects of industrial verification that make a theorem prover an effective tool for providing the overall framework for the verification task. Sections 5 thru 8 go into these aspects in more detail. We finally present conclusions and ideas for how industrial hardware verification can make progress.

2 Typical Doubts Concerning Theorem Proving

We begin our case for theorem proving by addressing two issues that have traditionally been seen as serious drawbacks to theorem proving. The first concern is that the logical language used in theorem provers is difficult for hardware engineers to understand and write. The second concern is that the task of proving theorems is too difficult or tedious for hardware engineers to undertake. As described below our experience with the industrial AAMP5 verification is that neither of these issues need be serious obstacles to the industrial us of theorem provers.

The first concern is that logic (higher-order or other) is too esoteric to be embraced or even understood by hardware engineers. This need not be the case for hardware verification. Our experience has been that hardware designs can be translated into a theorem prover in a very stylized manner that preserves the readability of the original HDL description.

The AAMP5 verification effort provides evidence for this view. In the AAMP5 effort the PVS specifications were subject to formal inspections. The hardware engineers found the specifications understandable. Additionally, the rigor of the specifications helped the engineers discover bugs in the AAMP5 design [14]. Eventually the hardware engineers were able to use the PVS specification language on their own.

Additionally, some of the more esoteric features of the specification language were effectively used. Those features contributed to the specification of the general framework to be discussed in Section 5 as well as for the maintainance of a reusable infrastructure. For example a general bit-vector library was specified and made extensive use of in a variety of circuits.

The other main concern is that theorem proving is too low-level, time consuming, and demanding a task to be performed by a hardware engineer. This concern is addressed by incorporating efficient special purpose verification tools into the theorem prover. These tools along with powerful low-level inference steps and an extensible strategy language can relieve the hardware engineer from the overly mundane and overly demanding tasks required in many theorem provers [7, 16].

Again, the AAMP5 and its sequel the AAMP-FV verification [14] efforts bears this out. The initial proofs of several instructions were performed at SRI to develop the proof infrastructure needed to automate the process. These proofs were then presented to hardware engineers at Rockwell-Collins who were then able to carry on proving the correctness of additional instructions on their own. Of course there will remain proofs that are going to be difficult for non-experts to do, but we believe that for most hardware proofs stylized approaches can be developed that hardware engineers will be able to follow.

3 Hardware Verification

In the 1980s Gordon [9] started the field of hardware verification. While not yet achieved, we believe that the practical industrial use of hardware verification is only a matter of time. We hold this belief in spite of the past failure of software verification to transition from research to industry. Our optimism is based on the nature of hardware and important differences between hardware and software.

The most obvious difference between software and hardware is the size of the task. One of the major obstacles to software verification is the shear enormity of the task. Clearly the size of even the most complex hardware circuits is considerably smaller than that of the typical software. The relative small size of hardware has allowed state space exploration tools such as model-checkers to be effective at verifying realistic hardware circuits[1].

More important than size, however, is the nature of the verification task itself. It has not been possible to separate software verification from the verification of the underlying algorithms. This makes software verification very often a sophisticated intellectual task that requires intelligent user guidance. Even when the proof is not that sophisticated it still involves subtle user direction to instantiate quantifiers, etc.

Hardware verification, on the other hand, is generally shallow, allowing automatic state-space enumeration, language containment, or symbolic rewriting methods to be used successfully. Even when intelligent human interaction is needed this interaction can very often be isolated to a few key choices that can be presented to the user in a predictable manner. Typically, the complexity of a hardware verification task is not the *first-order logic* search for quantifier instantiations, but the *propositional logic* problem of performing a large case-analysis quickly. The large amount of repeated structure in hardware verification problems has made techniques such as BDDs useful for efficiently performing this case analysis.

Even when the verification of a specific circuit does not lend itself to complete automatic analysis by a decision procedure, very often the verification problem can be broken down in a stylized manner to decidable subproblems. These can then be attacked using automatic tools. Additionally, when the circuit is too large to be fully verified there are natural abstractions and simplifications that can be made that allow automatic verification of a simpler problem that can give a greater degree of confidence in the correctness of the original circuit. Theorem provers are well suited for thee types of activities.

Finally, just as Hoare-triples or Floyd-inductive assertions provide a general framework for program verification, the verification of hardware can often be stated in a larger more general framework that can act as a guide for the larger verification effort [8].

4 Industrial Verification

Hardware verification in industry has several characteristics and needs that distinguish it from the type of verification done by researchers. In this section we enumerate these criteria and show how an interactive theorem prover can meet those needs.

1. Industrial hardware design is characterized by fast design cycles and rapid technological changes.

   This rapid change is likely to make any single verification technology or framework obsolete before it can be scaled up to be successfully applied to an industrial design. Simulation and CAD tools get around this problem by providing a very general customizable framework with a flexible hardware design language. An interactive general-purpose theorem prover can also provide such a flexible environment to support the rapid technology changes in industry.

2. Rapid design changes make efficient reuse of existing specifications, hardware descriptions and verification proofs very important.

   Expressive specification languages and proof strategy languages help achieve greater reuse.

3. Large industrial designs are typically made up of diverse components that work together to achieve the intended behavior.

   This makes it unlikely that any one verification technology is going to be useful for the entire verification effort. For example, a microprocessor consists of a memory unit, ALU, cache, pipelined data-paths, etc. The subcomponents might be best verified using different tools. For example, model-checking would be appropriate for verifying the correctness of a cache-coherence protocol, while rewriting and symbolic simulation would be appropriate for verifying the correctness of a micro-controller, and more specialized BDD-based tools would be effective in verifying the ALU. Additionally, the verification of the interaction of the different components might use still another method. A general-purpose theorem prover that has access to many decision procedures can best manage the integration of all these verification sub-tasks as well as perform some of the verification it self.

4. Industrial hardware designs can be very large and complex.

   This makes verification of these designs at different levels of abstraction important. A general-purpose theorem prover can provide a means for performing verification at the specification level as well as at the implementation level. Additionally, the hierarchical decomposition of the verification task can be managed by the theorem prover.

   The large size of industrial designs also makes partial verification important to quickly find and detect bugs. A theorem prover can aid in this task in several ways. One is to perform simulation and partial verification directly in the theorem prover by sufficiently constraining the design. Another is to interface to other simulation and CAD tools or model-checkers. The advantage of using a theorem prover as an intermediary is that only one specification or design need be used, instead of having to worry about the consistency of different forms of the same description.

   This brings us to our last criteria:

5. Documentation is very important in industry.

   As noted above the theorem prover can provide a single specification that can be used by the different verification as well as CAD tools.

[1] This technology has also been effective at verifying protocols, a specific type of software.

We now illustrate these points and show how a theorem prover can make use of them to perform large scale industrial verification. These points are generally applicable to any state of the art theorem prover with an expressive specification language and effective theorem proving support. We use PVS to concretely illustrate these points.

5 General Framework

One advantage of using a higher-order logic is that it is expressive enough to state the high-level basis for correctness appropriate to a wide class of circuits. In model-checking or special purpose tools a specific framework is built in to the tool. In a higher order prover an appropriate framework can be specified for any given task.

In this section we describe one approach for specifying the high level correctness of synchronous sequential circuits such as microprocessors. Other frameworks are discussed in [1, 8, 20].

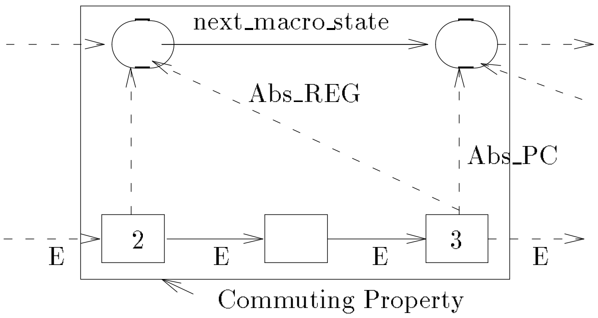

Figure 1: Pipelined Microprocessor Correctness

Formal microprocessor verification involves specifying the processor as a machine that interprets the instructions in the instruction set at two levels—*macro* and *micro*. The macro-machine gives a programmer's view of the processor, hiding internal details such as pipelining. The micro-machine describes the processor at a lower level, such as the architecture or register-transfer levels.

The correctness criterion for a pipelined processor is expressed, as shown in Figure 1, as a desired correspondence between the execution traces of the macro and micro-machines. In the figure, `E` and `next_macro_state` denote the micro-machine and macro-machine state transition functions, respectively. The objective of the correctness criterion is to ensure that the micro-machine does not introduce any behaviors not allowed by the macro-machine. Since the micro-machine usually takes a number of cycles to execute an instruction that is viewed atomically at the macro level, the two machine traces are required to correspond only at selected points (called *visible* states) in the micro-machine trace. For example, a possible definition of a visible state is one in which an instruction gets completed and a new instruction begins. A visible state is shown in Figure 1 with the number of the current instruction in that state. Since macro-state is an abstraction of micro-state, it is necessary to define an *abstraction* function `ABS` (shown separately as `Abs_PC` and `Abs_REG` in the figure for reasons that will be explained shortly) that returns the macro-state corresponding to the state of the micro-machine at any given time.

Given the notions of `ABS` and visible state, the correctness criterion described by Figure 1 can be stated as follows[2]: "Every micro-machine trace starting with an initial state (`S0`) that abstracts to a macro-state (`s0`) must be abstractable to the macro-machine trace starting with the initial state `s0`." To prove such a correspondence between two infinite traces, it is sufficient to prove the *commuting* property that captures the correspondence between the traces for an arbitrary step of the macro-machine, i.e., between two arbitrary, but successive visible states.

```
traces [state: TYPE]: THEORY

BEGIN
  i, n: VAR nat
  next_type: TYPE = function[state -> state]
  next: VAR next_type
  s: VAR state

  next_to_the_n (next, s, n): RECURSIVE state =
    IF n = 0 THEN s
       ELSE next_to_the_n(next, next(s), n-1)
    ENDIF
  MEASURE (LAMBDA next, s, n: n)

  trace(next, s): sequence[state] =
     (LAMBDA i: next_to_the_n(next, s, i))
END traces
```

Figure 2: Theory `traces`

Figures 2 and 3 show sample PVS theories that capture this definition of correctness. Figure 2 defines a generic theory of `traces` as a sequence of states, such that state $i+1$ is obtained by calling the next state function on state i. In Figure 3 the theory `trace_equiv` defines the notion of trace equivalence described in the previous paragraph. The theorem `trace_equiv` can be proven from the assumption `Commutes`. When this theory is instantiated for a particular design it will generate the `commutes` obligation for that design as a verification condition.

The size and clarity of these two theories compare very favorably with that of a lisp or C++ implementation of the same ideas in a stand-alone tool. Additionally, as microprocessors become more complicated the statement of correctness may need to be altered. Adding to or changing specifications such as those in Figures 2 and 3 is much easier and straightforward than either changing the implementation of a special purpose verification tool or switching to a new tool.

For simple pipelined machines the above theory can be used as a framework for correctness. In a pipelined

[2] We also need to ensure that processor resetting will always lead to an initial state in which the correspondence required by `ABS` holds.

```
trace_equiv [Astate: TYPE, A: [Astate -> Astate],
             Istate: TYPE, I: [Istate -> Istate],
             Abst: (surjective?[Istate, Astate])]: THEORY
BEGIN

ASSUMING

  Commutes: ASSUMPTION
     FORALL (s: Istate): Abst(I(s)) = A(Abst(s))

ENDASSUMING

IMPORTING traces[Astate], traces[Istate]

  r : VAR Astate
  s : VAR Istate

  Atrace(r): sequence[Astate] =
    traces[Astate].trace(A, r)

  Itrace(s): sequence[Istate] =
    traces[Istate].trace(I, s)

  trace_equiv: THEOREM map(Abst)(Itrace(s)) = Atrace(Abst(s))

END trace_equiv
```

Figure 3: Theory `trace_equiv`

processor a new instruction enters the pipeline before the results of the previous uncompleted instructions are at their destinations. This characteristic of pipelining is handled by allowing `ABS` to peek into future micro-states, possibly non-visible, following the current visible state (eg., `Abs_REG` shown in Figure 1) to get correct values for some of the macro-state components. Typically, the values for all but the program counter must be obtained from the future state. This can be done without modifying the above framework. All that is required is that the definition of `ABS` be appropriately specified.

Also, some pipelined processors, typically RISC processors, do not inhibit all kinds of pipeline hazards. For example, they allow instructions following a conditional branch to execute assuming the branch would not be taken leaving the burden of generating correct code to the compilers. One way to handle this is (as shown in [19]) either to restrict the set of programs for which the two machines are related or expose some of the effects of pipelining in the macro-machine. Again this could be done very straightforwardly in a theorem prover even without changing the overall framework. However adjustments to the general framework might be needed for super scalar processors [1].

Automated tools have been built and used for simple pipelined processors [2,4], and they can deal with many of the above issues. However, one complication that would create problems for an automatic tool is when the execution time for a single instruction (the distance between visible states) is indefinite and varying and depends on the liveness of some other part of the circuit. For example, in the AAMP5 there is a separate instruction pre-fetch unit which results in instruction fetches and data transfers happening independently with respect to the interpretation of instructions in the pipeline. In this case induction is needed to prove the correctness of some instructions and thus, the automated methods would not be applicable. However, there are sub-procedures of those automated methods such as the efficient simplification of IF-THEN-ELSEs that would still be useful. The logical framework is best specified in a theorem prover. The lower level tasks are best implemented in automatic procedures.

This section has illustrated one of the main advantages of a general-purpose theorem prover over more specialized tools. The theorem prover can specify different frameworks precisely and efficiently without having to re-implement a system. We now explore how to move from this high level framework to the low level proofs that make up a complete verification.

6 Reuse

The theories describing the general verification framework illustrates one aspect of reuse. Those theories are very general and can be instantiated with particular abstraction functions, processor state types, and next-state functions appropriate for very different types of circuits. Other examples of specification reuse is the n-bit word library that is part of the AAMP5 specification and the IEEE floating point specification done in PVS [15]. The very expressive high-level type system and specification language typical of interactive theorem provers make this type of reuse possible.

Specification reuse is by no means unique to theorem provers. Code reuse is also obtained with many object-oriented languages and might very well be part of more automated tools. Another type of reuse that is more specific to theorem provers is *proof* reuse. When carrying out hardware proofs very often similar proofs are used for many of the sub-verification tasks. A theorem prover with an extensible proof-strategy language can be used to build reusable generic proof commands for these verification sub-tasks. In fact one can view automatic tools as an extremely efficient theorem prover strategy that is applicable to a restricted, but sizable class of tasks.

In the AAMP5 verification effort several cliched proof attempts were identified. This greatly assisted the hardware engineer in being able to do much of the verification with out the aid of a theorem proving expert.

7 Inter-Operation of Verification Tools

As described above automatic verification tools can be viewed as special purpose proof strategies in an interactive theorem prover. Inorder to be useful for practical verification these proof strategies must be as efficient as possible [7]. PVS already incorporates many different decision procedures. These include a BDD based propositional simplifier, combined ground decision procedures for equality, arithmetic and reading and writing memory, a conditional rewriter which effectively uses the combined ground decision procedures, and a propositional temporal logic model-checker.

Additionally PVS includes many defined strategies that are generally used as black-boxes. Perhaps the most important of these is rewriting with respect to a equations in a theory.

A complete verification is most *efficiently* handled by using a combination of these procedures. A verification engineer relying solely on the theorem prover or on any one automated procedure will be much less productive.

7.1 Using a Model Checker

Model-checking provides a fully automatic technique for deciding an important class of verification problems, such as liveness properties of finite state systems. The importance of such automation cannot be overemphasized. The effort required to construct logical proofs of correctness for these problems in a theorem prover can be large and tedious since it requires the discovery of suitably strong invariant and possibly induction. The primary disadvantage of model-checking is that it only works well for small state spaces. This limitation can be partially overcome through the use of Binary Decision Diagram (BDD) and symbolic model-checking.

Even so, the state explosion problem limits the applicability of model-checking in practice. However, in practice, although an entire system is too large for a model-checker to handle it is possible to identify parts of the system or certain important properties of the system that can be proved automatically using a model-checker on a suitable abstraction of the complete system. For example, the AAMP5 processor design employs, besides the main data-path, separate semi-autonomous subunits, the Look-ahead Fetch Unit (LFU) to pre-fetch instructions and the Bus Interface Unit (BIU) to arbitrate requests for data and instruction fetch requests to the memory. In the AAMP5 verification, we abstracted parts of the LFU/BIU by assuming that the LFU/BIU ensemble satisfied a set of liveness properties since it was quite tedious to verify them in a theorem prover. For example, we assume that if the data-path of the AAMP5 is "held" because the data bus is being controlled by the LFU, then eventually the system will reach a state in which the data hold is released. We verified the microcode correctness using such properties about the LFU/BIU.

These liveness properties can be proved automatically using a model-checker on a finite state model of the LFU/BIU design than using a theorem prover. If we had the capability to invoke a model-checker as a primitive proof step within a theorem prover, then we could discharge the liveness properties using the model-checker after defining an appropriate abstraction function on the LFU/BIU within the theorem prover. We have implemented such an integration between a model-checker and PVS and tested it to verify [17] a simplified version of the LFU/BIU model encountered in the AAMP5 design. We plan to extend it to verify the LFU/BIU parts of the AAMP5.

Another example of the use of a model-checker in a theorem prover is in the verification of Peterson's n-process mutual exclusion protocol [18]. The verification is by induction on the number of processes. The model-checker is able to automatically verify the correctness of the base case (one process instance) and the two process instance of the protocol. The theorem prover does the induction and uses abstraction to reduce the inductive step to the two process case which the model-checker can quickly decide.

7.2 Other Decidable Theories

Other aspects of hardware verification are amenable to different automated tools. All of [4, 6, 7] describe decidable theories or strategies that effectively deal with verification of the data flow through a pipelined microprocessor. The essence of these methods is to use rewriting to generate large IF-THEN-ELSE expressions that then have to be compared for equality. This comparison is done by performing a combinations of case analysis with decision procedures for ground equality and other decidable theories. An overall correctness criteria is built into the tool described in [4]. As argued above a better approach would be to separate the correctness criteria and decision procedure. The correctness criteria may not remain universally useful. The decision procedure part which decides the equality of large IF-THEN-ELSE expression will be useful in a wide variety of situations.

Thus not only do efficient decision procedures enhance the capabilities of the theorem prover, but the theorem prover can make narrow decision procedures more generally applicable.

8 Dealing With Size

The large size of industrial circuits also raises important issues. One method for dealing with a large verification problem is to break the verification task into smaller subtasks. As indicated above this is something that theorem provers do naturally. Another way is to verify less than the entire circuit. With a stand-alone tool this often involves rewriting the circuit specification. A problem with this approach is that the relationship between the simpler verified circuit and original circuit may not be clear and, perhaps worse, may not be documented.

A theorem prover naturally allows partial verification to be done in the same context as the original verification. This is accomplished not by rewriting the circuits but by further constraining the original circuit. For example, one can add as an hypothesis that the interrupt signals remain low in order to verify the correctness of the circuit without dealing with the added complexity of interrupts. Taking this approach to the extreme one can restrict the conditions enough so that the theorem prover is essentially operating as a simulator.

Another approach to dealing with large complex circuits is to hierarchically decompose them. This involves verifying circuits that are of intermediate complexity between the most abstract specification and lowest level circuit. An example would be verifying a pipelined processor with respect to an un-pipelined processor and then verifying the un-pipelined processor with respect to the instruction set architecture.

One final way of dealing with size is to interface to other CAD tools. If the circuit specification can be translated to an HDL of a simulator then the early stages of design and verification can be undertaken by the traditional simulation process.

9 Conclusions

While theorem proving has been viewed as an esoteric method unsuited for real industrial verification, we argue that in fact the opposite is true. A general-purpose theorem prover with the right capabilities is a practical tool for real hardware engineers to gain greater assurance that their circuits are correct. The automated tasks of simulation, model-checking, symbolic evaluation can be effectively embedded in a general-purpose theorem prover so that the hardware engineer has at his disposal effective tools for the more brute force aspects of the verification task. At the same time, those tasks that are not easily automated can at least be specified and attacked by the hardware engineer. Even if he is not capable of handling them, the theorem prover still provides feedback as to which parts of the design have not been covered.

Furthermore, while any one automated tool might be useful at some aspects of industrial verification, only a general-purpose theorem prover with access to automated tools effectively addresses all those aspects.

These conclusions are not just based on argument, but have also been born out by an actual industrial verification effort involving the AAMP5 processor [14]. This additionally provides a model for future research. Research in increasing the capabilities of automated tools and interactive theorem provers is important, as is research aimed at enhancing their integration. But, perhaps equally important is research that actually applies this combination to actual industrial examples. Only through this type of research will the type of interactions and the requirements on the system be fleshed out enough to make progress toward long term industrial use of hardware verification.

10 Acknowledgements

We thank John Rushby and Sree Rajan for helpful comments on the paper.

References

[1] Mark Aargard and Miriam Leeser. Reasoning about pipelines with structural hazards. In Kumar and Kropf [13], pages 15–34.

[2] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proceedings of the 31st Design Automation Conference*, pages 596–602. Association for Computing Machinery, June 1994.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Stanford, CA, June 1994. Springer-Verlag.

[5] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *19th ACM Symposium on Principles of Programming Languages*, pages 343–354, Albuquerque, NM, January 1992. Association for Computing Machinery.

[6] D. Cyrluk and P. Narendran. Ground temporal logic—a logic for hardware verification. In David Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 247–259, Stanford, CA, June 1994. Springer-Verlag.

[7] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Kumar and Kropf [13], pages 203–222.

[8] David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[9] Mike Gordon. Proving a computer correct. Technical Report TR 42, University of Cambridge, Computer Laboratory, 1983.

[10] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in Systems Design*, 1(2/3):151–238, October 1992.

[11] W.A. Hunt. The mechanical verification of a microprocessor design. In *Proc. of IFIP Working Conference on From H.D.L Descriptions to Guaranteed Correct Circuit Designs*, 1986.

[12] J. Joyce, G. Birtwistle, and M. Gordon. Proving a computer correct in higher order logic. Technical Report 100, Computer Laboratory, University of Cambridge, 1986.

[13] Ramayya Kumar and Thomas Kropf, editors. *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

[14] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[15] Paul S. Miner. Defining the ieee-854 floating-point standard in pvs. NASA Technical Memorandum 110167, NASA Langley Research Center, June 1995.

[16] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[17] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, Lecture Notes in Computer Science, Liege, Belgium, June 1995. Springer-Verlag. To appear.

[18] N. Shankar. Computer-aided computing. In *Proceedings of the 3rd International Conference on the Mathematics of Program Construction*, 1995. (To appear).

[19] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[20] Phillip J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. In Kumar and Kropf [13], pages 33–51.