Linking BDD-Based Symbolic Evaluation to Interactive Theorem-Proving*

Jeffrey J. Joyce
Carl-Johan H. Seger
Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z2 Canada

Abstract – A novel approach to formal hardware verification results from the combination of symbolic trajectory evaluation and interactive theorem-proving. From symbolic trajectory evaluation we inherit a high degree of automation and accurate models of circuit behaviour and timing. From interactive theorem-proving we gain access to powerful mathematical tools such as induction and abstraction. We have prototyped a hybrid tool and used this tool to obtain verification results that could not be easily obtained with previously published techniques.

1. Introduction

Designing complex digital systems in VLSI technology usually involves working at several levels of abstraction, ranging from very high level behavioral specifications down to physical layout. One of the main difficulties in this process is to verify the consistency of the different levels of abstraction. Simulation is often used as the main tool for "checking" the consistency. Despite major simulation efforts, serious design errors often remain undetected. Consequently, there has been a growing interest in using formal methods to verify the correctness of designs. There are several approaches to formal hardware verification: theorem-proving, state machine analysis, and symbolic simulation to mention a few [11]. These methods all have strengths and weaknesses. In this paper we will illustrate how theorem-proving can be rigorously linked with symbolic simulation to gain a verification methodology that draws on the strengths of each approach. In particular, we have developed a novel approach to formal hardware verification which extends BDD-based symbolic simulation techniques with more general-purpose reasoning tools such as abstraction and mathematical induction. The result is a hybrid approach to formal hardware verification that offers considerable promise in bridging the current gap between conventional verification techniques, such as switch-level simulation, and more esoteric formal techniques. We have implemented a prototype tool for our hybrid approach and used this tool to derive verification results that would be difficult to achieve with previously published techniques.

*This paper will appear in the Proceedings of the 30th Design Automation Conference to be published by the IEEE. This research was supported by operating grants OGPO 109688 and OGPO 046196 from the Natural Sciences Research Council of Canada, by research contract 92-DJ-295 from the Semiconductor Research Corporation, and by fellowships from the Advanced Systems Institute.

Our hybrid approach can be seen as the latest step in a chain of evolution which began with the development of circuit models for switch-level simulation in the early 1980's [2]. Switch-level simulation is a commonly used verification technique supported by a number of commercial tools. The next step in this evolutionary chain was the development of symbolic switch-level simulation in the mid 1980's [3]. This symbolic approach to switch-level simulation is supported by tools such as the COSMOS system from Carnegie-Mellon University [4]. Symbolic simulation can be seen as an extension of ordinary switch-level simulation where node values may be treated symbolically, that is, variables may be used to represent node values instead of constants such as `T` and `F`. A symbolic simulator can be used to verify assertions about the state of a circuit that results from a given sequence of inputs – for instance, to show that the value of a particular output node is accurately described by a formula parameterized by a set of variables representing input values. Next in this evolutionary development was an extension to symbolic simulation that made it possible to verify assertions about state trajectories, that is, sequences of states rather just single states. In addition to treating node values symbolically, symbolic trajectory evaluation provides a rigorous technique for verifying temporal relationships between these node values. Recent versions of both COSMOS and a related system called Voss [12] from the University of British Columbia provide support for symbolic trajectory evaluation. Finally, our hybrid approach, as the latest step in this evolutionary chain, extends symbolic trajectory evaluation with a set of general-purpose reasoning tools. We have implemented a prototype software tool for our hybrid approach by means of an interface between the Voss system and an interactive theorem-prover called HOL from the University of Cambridge [10].

As an extension of symbolic trajectory evaluation, our hybrid approach has inherited the following strengths:

- In contrast to other formal approaches which often involve the use of over-simplified models, verification results obtained by means of our hybrid approach are based on models of circuit behaviour and timing which are widely used in conventional CAD.
- Unlike several other approaches to formal hardware verification which require extensive re-training before any useful results can be achieved, there is a relatively smooth learning curve which allows a novice to start using our hybrid

approach as a form of conventional switch-level simulation and then gradually acquire increasingly levels of expertise in the use of more esoteric techniques.

Our hybrid approach extends symbolic trajectory evaluation with the addition of more general-purpose reasoning tools such as abstraction and mathematical induction. With these additions, our hybrid approach acquires several new strengths including:

- Verification results obtained by means of symbolic trajectory evaluation can be formally related to higher levels of abstraction including the formal specification of mixed software/hardware systems.
- It provides a rigorous framework for combining results obtained by symbolic trajectory evaluation to yield verification results that, as a whole, would exceed the capacity of symbolic trajectory evaluation.
- We have a rigorous framework for developing mathematically sound interfaces between symbolic trajectory evaluation and conventional (as well as experimental) hardware description languages.

The motivation for combining symbolic trajectory evaluation with general-purpose reasoning tools is illustrated by the limitations of symbolic trajectory evaluation in verifying the design of a 32-bit multiplier. At best, BDD-based symbolic trajectory evaluation can be used to achieve a partial result expressed by a set of verification conditions. While these verification conditions collectively imply that a 32-bit multiplier actually does multiplication, this inference must be done informally outside the framework of symbolic trajectory evaluation. Any attempt to achieve a complete result by verifying an assertion expressed directly in terms of multiplication would exceed the capacity of BDD-based symbolic trajectory evaluation [7]. With the addition of general-purpose reasoning tools, our hybrid approach allows us to to complete a correctness proof for the multiplier within a rigorous framework by formally proving that the verification results obtained by symbolic trajectory evaluation logically imply that the multiplier really does multiplication. The use of general-purpose reasoning tools such as mathematical induction and abstraction are essential for this result.

The remaining sections of this paper describes the main elements of our hybrid approach and its applications. Section 2 outlines the role of symbolic trajectory evaluation in our hybrid approach. In Section 3 we explain how we extend our ability to verify circuits using symbolic trajectory evaluation with the more general-purpose reasoning tools by embedding symbolic trajectory evaluation inside the logic of an interactive theorem-prover. Section 4 describes infrastructure that we have developed to increase the usability of our hybrid approach. We present an example in Section 5 to illustrate the advantages of our hybrid approach. Finally, a summary of our work and outline of our plans for future development of this work are given in the Section 6.

2. Symbolic Trajectory Evaluation

Our prototype implementation of a hybrid verification tool includes the implementation of a symbolic trajectory evaluator called Voss. The Voss system can be used to verify assertions of the form,

```
FSM fsm (assumptions,conclusions)
```

where `fsm` denotes a finite-state machine and the pair `(assumptions,conclusions)` expresses a relationship over the trajectories of this finite-state machine. The finite-state machine denoted by `fsm` is specified by a set of next-state functions accessed by the Voss system from an external file. This finite-state machine is a behavioural model of a digital circuit which can be automatically generated from a transistor netlist by a separate tool called Anamos [4] or from a gate netlist in Silos format [15]. The parameters `assumptions` and `conclusions` each denotes a list of atomic constraints. If the above assertion is true for the finite-state machine denoted by `fsm`, then any trajectory of this finite-state machine which satisfies all of the atomic constraints in the `assumptions` list must also satisfy all of the constraints in the `conclusions` list. Each atomic constraint is a 5-tuples of the form `(b,n,v,s,f)` which, for a given trajectory, denotes the constraint that "if the Boolean expression `b` is true then the node named by `n` has the value `v` in all states of the trajectory from the start state `s` up to but not including the final state `f`".

To give a very simple example, the assertion,

```
FSM inverter (
  [(T,`input`,F,0,1)],
  [(T,`output`,T,1,2)])
```

expresses a relationship between the input and output node of an inverter where the output value is delayed by one time unit. We have used the constants `T` and `F` to denote node values. The verification of this assertion can be viewed as just an instance of ordinary switch-level simulation. The assumption constraint `(T,`input`,F,0,1)` causes the input node of the inverter to be set to `LO` while the conclusion constraint `(T,`output`,T,1,2)` checks that the output node becomes `HI` after the elapsed of one time unit. A slightly more sophisticated approach is illustrated by the assertion,

```
FSM inverter (
  [(T,`input`,v,0,1)],
  [(T,`output`,~v,1,2)])
```

where the constants `F` and `T` have been replaced by the symbolic expressions `v` and `~v`. The use of a variable for this purpose can be viewed as an instance of symbolic simulation. In this case, the assertion includes the possibility that the initial value of the input node is `T` in addition to the possibility that it is `F`.

It may appear that the temporal scope of the above assertion is limited to the first two instants of discrete time – that is, "if the input at time 0 is `v`, then the output at time 1 will be `~v`." However, the temporal scope of this assertion actually extends infinitely along every trajectory of the finite-state machine. This is because the automatic verification procedure considers every state of the finite-state machine to be a possible initial state of the machine. At any point along any trajectory, the current state corresponds to the initial state of some other trajectory. Therefore, verification that some property holds for every possible initial finite sub-sequence of states implies that the property holds for all finite sub-sequences (not just initial sub-sequences) of every trajectory. Because the temporal scope of the above assertion extends infinitely along every trajectory, the assertion can be accurately interpreted to express the property that "for all times `t`, if the value of the input node of the inverter is v, then the value of the output node at time t+1 will be ~v".

The specification language accepted by the Voss system includes a number of built-in constants and functions – for example, a set of built-in Boolean constants and operators including T (true), F (false), ∧ (conjunction), ∨ (disjunction) and ~ (negation). The Voss system also allows a number of constructs commonly found in functional programming languages such as Lisp and ML to be used to express assertions. For example, the assertion,

```
let xor (n,m) = (n ∧ ~m) ∨ (~n ∧ m) in
let sum (a,b,cin) = xor (xor (a,b),c) in
let cout (a,b,cin) =
   (a ∧ b) ∨ (a ∧ c) ∨ (b ∧ c) in
let assumptions =
   [(T,‘a‘,a,0,20);
    (T,‘b‘,b,0,20);
    (T,‘cin‘,cin,0,20)] in
let conclusions =
   [(T,‘sum‘,sum (a,b,cin),10,20);
    (T,‘cout‘,cout (a,b,cin),10,20)] in
FSM fulladder (assumptions,conclusions)
```

illustrates the use of a nested sequence of let-expressions to express a correctness property for the design of a full adder. The assumption part of the above assertions specifies the constraint that the three inputs nodes, ‘a‘, ‘b‘ and ‘cin‘ have constant values denoted by the variables a, b and cin from time 0 until time 20. The conclusion part specifies the constraint that the output nodes ‘sum‘ and ‘cout‘ will have the constant values denoted by the variables sum (a,b,cin) and cout (a,b,cin) respectively. The notational convenience of using a let-expression to define the xor function allows us to express this property in a relatively succinct manner.

With the expressive power of these functional programming language constructs, it is possible to write down assertions of considerable complexity and use the Voss system to automatically verify these assertions with respect to a finite-state machine derived from the design of a digital circuit. We have used Voss in this manner to automatically verify several non-trivial designs including a 32-bit version of the Tamarack-3 microprocessor [9] and a 32-bit wide 4 stage pipelined integer unit with a dual-ported register file with 32 general purpose registers [1].

3. Linking Voss to an Interactive Theorem-Prover

Our prototype implementation of a hybrid verification tool is based on an interface between the Voss system and an interactive theorem-prover called HOL [10]. This interface is more than the *ad hoc* translation of output from one verification tool into input for another verification tool. A considerable amount of our development effort has focused on the establishment of a "mathematical interface" between symbolic trajectory evaluation and interactive theorem-proving as a sound foundation for the development of a tool interface. This paper focuses on the tool interface since the details of the mathematical interface are beyond the scope of our discussion here; details of the mathematical interface may be found in a technical report [13] that presents a more theoretical view of our hybrid approach to formal hardware verification.

The cornerstone of our hybrid approach is the definition of several new predicates in the HOL system which establish a mathematical link between the specification language of the Voss system and the specification language of the HOL system. This includes the formal definition of the predicate FSM mentioned in the previous section. In HOL jargon, the establishment of this link can be described as a "semantic embedding" of Voss within higher-order logic (the specification language of the HOL system). The establishment of this mathematical link causes the specification language of Voss to become a subset of the specification language of the HOL system.

In addition to this minor extension of the HOL specification language to include Voss assertions, we extend the set of built-in proof procedures of the HOL system with a new proof procedure based on symbolic trajectory evaluation. This new proof procedure is implemented as a remote function call to the Voss system which is run as a child process of the HOL system. This new proof procedure, called VOSS_TAC, is invoked with a single argument – namely, a Voss assertion expressed as a term of higher-order logic. The assertion is passed directly to the Voss system which uses symbolic trajectory evaluation to decide whether the assertion is true. If the assertion is successfully verified, the result T ("true") is returned to the HOL system and the assertion is transformed into a theorem.

Once an assertion has been verified by symbolic trajectory evaluation and transformed into a HOL theorem, this verification result may be used by the more general-purpose reasoning tools of the HOL system to derive additional verification results. For example, reasoning tools such as mathematical induction and abstraction might be used to combine a set of verification results obtained by means of VOSS_TAC into a single verification result. These tools might also be used to derive a more abstract verification result – for example, the specification of a software function in a mixed hardware/software system – from lower level verification results obtained by means of VOSS_TAC.

The extent to which these additional proof procedures are used depends on the expertise of the user. A novice user will probably begin by only using our hybrid verification tool as a symbolic simulator. A slightly more advanced user can use our tool for symbolic trajectory evaluation to verify assertions about the temporal relationships between states. Gradually a user may begin using other HOL proof procedures besides VOSS_TAC. At this skill level, a user should already be capable of using VOSS_TAC to achieve useful verification results. Therefore, interactive theorem-proving skills can be gradually acquired while useful work is being done with just symbolic trajectory evaluation.

4. Proof Infrastructure

In the development of our hybrid approach we have spent considerable effort on the development of proof infrastructure which increases the usability of our approach. In particular, our efforts to date have focused on the development of three main kinds of infrastructure:

- a library of arithmetic and logical operations on bitvectors,

- a very simple, experimental language called HCL for writing more succinct specifications, and
- general proof procedures for common verification tasks in the HOL-Voss system.

4.1. Arithmetic and Logical Operations

Correctness assertions about hardware designs frequently involves assertions expressed in terms of arithmetic and logical relationships. To provide support for these kinds of specifications, we have developed a library of arithmetic and logical operations on bit-vectors. For example, this library includes the infix operation `bvplus` that forms the result of adding two vectors of Boolean values. This operation could be used, for instance, to specify the correctness property that the output value of a 32-bit adder, denoted by the variable `sum`, corresponds to the result of applying the operation `bvplus` to the values of the two input vectors denoted by the variables `a` and `b`. The operation `sized` is used here to truncate or pad the result of the `bvplus` to make it into a 32-bit vector.

```
sum = (sized 32 (a bvplus b))
```

All of the arithmetic and logical operations in this library are formally defined as functions in the specification language of the HOL system. This is important because it allows us to formally validate these definitions. In particular, we have used the HOL system to formally derive correctness properties for these bitvector operations with respect to an interpretation of bitvectors (as a little-endian unsigned binary representation) and the standard notion of arithmetic based on Peano's axioms. For example, we have used the HOL system to prove the theorem,

```
∀a b. (bv2num (a bvplus b)) =
      ((bv2num a) + (bv2num b))
```

where `bv2num` is an abstraction function that converts a vector of Boolean values into a natural number. The above theorem establishes a rigorous correspondence between the bit vector operation `bvplus` and + where + is an arithmetic operation defined on natural numbers. There are two important points to make here. First, the library of bitvector functions is developed only once and the correctness proofs, carried out in the HOL system, are only done once. Secondly, since the Voss system will in fact use the same definitions during its execution, there is a very rigorous link between the arithmetic relations used in the HOL system to describe the specification and the actual bitvector versions used by the Voss system.

4.2. HCL - Higher-level Constraint Language

Earlier examples in this paper have shown how assertions can be expressed in terms of lists of 5-tuples of the form `(b,n,v,s,f)`. As shown previously with the example specification of the fulladder in Section 2, various constructs in the specification language such as `let`-expressions can be used to make assertions more succinct. However, in general, the use of 5-tuples to write down assertions, even with the help of various built-in language constructs, is far too cumbersome.

An elegant solution to this problem is to use the expressive power of the specification language to introduce a user-defined specification language that can be "compiled" into the 5-tuple format required by VOSS to perform symbolic trajectory evaluation. This approach involves four main steps. First, the syntax of the user-defined specification language is introduced as a new data type in the underlying logical framework of the HOL system. Secondly, we define a compiler function that compiles the user-defined specification language into the 5-tuple format. Thirdly, we formally specify the semantics of the user-defined specification language. The fourth and final step is to formally prove that the definition of the compiler function is correct with respect to the formal semantics of this language.

We have demonstrated this approach with the development of a very simple, experimental language called HCL. This user-defined language consists of a number of constructs for specifying waveforms – that is, for specifying temporal relationships between nodes and vectors of nodes. For example, HCL includes the infix operator `is` (`isv`) for expressing the instantaneous constraint that a particular node (vector of nodes) is equal to some Boolean value (vector of Boolean values). Another infix HCL operator, `during`, is used to express the temporal constraint that an instantaneous constraint holds during some specified interval. The HCL operator `and` can be used to combine temporal constraints. The following fragment of HCL, which appears again in Section 5, specifies the constraint that the node denoted by `phi` is false from time 0 until time 100 and then true until time 200. This fragment also specifies the constraint that two node vectors, denoted by the constants `Na` and `Nb`, are equal to the 16-bit, little-endian, unsigned binary representation of two natural numbers, `a` and `b`, from time 80 until time 200.

```
(((phi is F) during (0,100)) and
 ((phi is T) during (100,200)) and
 ((Na isv (sized 16 (num2bv a)))
  during (80,200)) and
 ((Nb isv (sized 16 (num2bv b)))
  during (80,200)))
```

After introducing the syntax of HCL as a new data type in the HOL logic, we formally defined a compiling function `CompileHCL` as a mapping from HCL to the 5-tuple format required for symbolic trajectory evaluation. For instance, the above fragment of HCL is mapped by `CompileHCL` to a list of thirty-four 5-tuples. Next, we defined a semantics function `SemanticsHCL` as a mapping from HCL to a predicate on trajectories of finite-state machines. The final step in this development was to prove that the definition of `CompileHCL` is correct with respect to the semantics of HCL as given by `SemanticsHCL`. Intuitively, this result states that the set of trajectories that satisfies a given HCL specification in terms of its formally defined semantics is identical to the set of trajectories that satisfies the set of 5-tuples that results when `CompileHCL` is applied to that particular HCL specification. The definitions of `CompileHCL` and `SemanticsHCL` along with a precise statement of the compiler correctness result may be found in a more theoretical presentation of our work [13].

There are several important points to emphasize here. First, the compiler function, `CompileHCL`, is defined in the HOL system. This allows us to reason about this definition – in particular, to show that it is correct with respect to the formal semantics of HCL. Secondly, the HOL definitions of `CompileHCL` is used directly by the Voss system to compile HCL specifications into the 5-tuple format. Our approach does not allow the possibility of mistakes that might occur in the hand-translation of a compiler specification into an implementation of the compiler specification. Thirdly, the task of formally verifying the compiler function definition needs to be done only once. Once this task has been completed, this correctness result becomes part of the supporting infrastructure of our hybrid approach. This correctness result is required when verification results obtained by means of symbolic trajectory evaluation are combined at higher levels. Fourthly, we note that the compiler correctness exercise is valuable in itself. In the course of verifying the definition of `CompileHCL`, we actually encountered a bug in the definitions of the compiler functions; consequently we feel the exercise was well worth the effort. Finally, we emphasize that this development of HCL is not intended to be yet-another hardware specification language. Instead, HCL is intended mainly to serve as an illustration of how a mathematically sound link can be developed between a specification language and the HOL-Voss system. Although HCL is a very simple language, we expect that it may be possible to use these same principles to develop a HOL-Voss interface for a more sophisticated hardware specification language – assuming that the language has a rigorously defined semantics.

4.3. General Proof Infrastructure

Finally, much of the work we have done is aimed at developing re-usable proof procedures that are commonly encountered during a typical HOL-Voss proof. It is interesting to note that many of the result in this set of proof procedures generalizes informal reasoning carried out manually before. Since all of these results have been formally proven in the HOL system the level of confidence is significantly increased.

5. An Example

In this section we illustrate the advantages of our two-level verification system in achieving verification results that would be difficult or impossible to achieve using either an approach based exclusively on theorem-proving or an approach based exclusively on symbolic trajectory evaluation. Our example is based on a Domino CMOS circuit with two 16-bit inputs, `[‘a.0‘;‘a.1‘;...;‘a.15‘]` and `[‘b.0‘;‘b.1‘;...;‘b.15‘]`, and one output bit `‘out‘`. The circuit design uses quite complex electrical phenomena and critical timing; this means that a rather sophisticated switch-level and delay model is needed to explain the operation of the circuit. The circuit is intended to compare the number presented on input `a` with the number presented on input `b`, both viewed as 16-bit, little-endian, unsigned binary representations, and produce an output `T` if and only if `a` $>$ `b` and `b` > 0. Since we would like to minimize the semantic gap between this intuitive notion of what we believe the circuit is supposed to do and the formal specification for the circuit, we require that the specification should be stated in terms of an arithmetic relation rather than a relation on bitvectors.

Clearly, the desired verification result cannot be achieved using symbolic trajectory evaluation exclusively since symbolic trajectory evaluation can only be used to directly verify assertions expressed in terms of bitvector operations – not arithmetic relations. As for a theorem-proving approach, it is possible, in principle, that this verification result could be achieved exclusively by means of interactive theorem-proving techniques. However, we are extremely doubtful that this result could be actually achieved in practice because there has never been a convincing demonstration of the ability of theorem-proving techniques to deal with complex electrical phenomena and timing behaviour for a non-trivial circuit.

With our hybrid approach, we have the best of both worlds. From symbolic trajectory evaluation our approach inherits accurate models of circuit behaviour and timing. From interactive theorem-proving we gain access to the ability to formally relate a bitvector level specification of this circuit to more abstract specification expressed in terms of an arithmetic relation. For these reasons, this particular verification problem is an excellent illustration of the unique advantages of the HOL-Voss system. We have used the HOL-Voss system to formally derive the following theorem which expresses the desired verification result for the "a $>$ b $>$ 0" circuit.

```
let phi = ‘phi‘ in
let Na = node_vec 16 ‘a‘ in
let Nb = node_vec 16 ‘b‘ in
let out = ‘out‘ in
forall a b::(0,65535).
  let assumptions =
    CompileHCL (
      ((phi is F) during (0,100)) and
      ((phi is T) during (100,200)) and
      ((Na isv (sized 16 (num2bv a)))
       during (80,200)) and
      ((Nb isv (sized 16 (num2bv b)))
       during (80,200))) in
  let conclusions =
     CompileHCL (
       (out is (a > b ∧ b > 0))
       during (160,200)) in
  FSM agrb16 (assumptions,conclusions)
```

The above theorem states the "a $>$ b $>$ 0" circuit, as represented by the constant `agrb16`, correctly compares the two 16-bit inputs for all input values between 0 and 65,535. The correctness property is expressed explicitly in terms of $>$ – an arithmetic operation rather than an operation on bitvectors. The HOL-Voss proof script (a sequence of commands called "tactics") required to generate a proof of the above theorem consists of only four very routine user-interaction steps. Execution of this proof script takes about half a minute on a NeXT Station (25MHz 68040 processor with 20M). Most of this time is in fact spent loading the necessary libraries. If this same result could ever be achieved exclusively by means of interactive theorem-proving (which would surprise us), we speculate that our four line proof script might compare to hundreds (and probably thousands) of user-interaction steps requiring hundreds of person-hours of interaction with the theorem-prover.

For the purposes of illustration, the above specification of the "a $>$ b $>$ 0" circuit only uses a small amount of the infrastructure that we have developed for HOL-Voss. With more extensive use of this infrastructure, it is possible to write more sophisticated specifications of this circuit in a form that is more convenient for combining with other verification results – however, these details lie beyond the scope of this paper.

In addition to the example described here, we have used HOL-Voss to verify several other circuits including a BCD (Binary Coded Decimal) converter and an 8-bit version of the Tamarack-3 microprocessor [9].

6. Related Work, Conclusions and Future Work

We believe that this work represents one of the first successful attempts to rigorously combine two different approaches to formal hardware verification. We are aware of previously published work done at IMEC in Belgium [8] on multi-level verification which shares a common goal with our approach in exporting verification results obtained by BDD-based methods to higher level verification tools. Distinguishing features of our approach include our emphasis on the establishment of a mathematical link between BDD-based methods and the underlying logical framework of higher level verification tools. Also, the two-level verification tool described in [8] relies heavily on a specific design methodology in order to automate much of the proof obligations. Our goal is to develop a very general tool in which the integrity of the proof is of major importance.

Our short-term development efforts will concentrate on the development of more infrastructure to increase the usability of our approach. Our goal is to minimize the amount of interactive theorem-proving expertise required to achieve significant verifications results. Our current implementation of HOL-Voss is presented to the user as an extension of the HOL system where symbolic trajectory evaluation is made available as an additional proof procedure called `VOSS_TAC`. We are now considering a new approach where HOL-Voss might be split into two separate tools. One of these tools would be an extension to the HOL system intended for use by theorem-proving experts for developing infrastructure such as interfaces to new specification languages. The second tool would be an extension to the Voss system intended for use by CAD designers for verifying circuit designs. In this "front-room, back-room" paradigm, infrastructure developed by the theorem-proving experts using the first tool would be used to enhance the functionality of the second tool.

In summary, we believe that our hybrid approach offers considerable promise as a practical verification methodology that could bridge the current gap between conventional CAD practice and formal hardware verification techniques that have evolved over the past 10-15 years. We also believe that our hybrid approach will serve as a prototypical model of how other verification techniques can be combined – for instance, the combination of model-checking and interactive theorem-proving.

References

[1] D. Beatty, R.E. Bryant, and C-J. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation", *IEEE ACM Design Automation Conference*, San Francisco, CA, June 1991.

[2] R.E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. on Computers* Vol. C-33, No. 2, February, 1984, pp. 160–177.

[3] R.E. Bryant, "Symbolic Verification of MOS Circuits", *1985 Chapel Hill Conference on VLSI*, May, 1985, pp. 419-438.

[4] R.E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits", *24th Design Automation Conference*, June 1987, pp. 9-16.

[5] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation" *IEEE Transactions on Computers*, Vol. C-35, No. 8, December 1986, pp. 677–691.

[6] R.E. Bryant, and C-J. Seger, "Formal Verification of Digital Circuits Using Symbolic Ternary System Models", in *Computer-Aided Verification '90*, Procs. of a DIMACS Workshop, American Mathematical Society, 1990, pages 121-146.

[7] R.E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Applications to Integer Multiplication", *IEEE Transactions on Computers*, Vol. C-40, No. 2, February 1991.

[8] M. Genoe, L. Claesen, E. Verlind, F. Proesmans, and H. De Man, Illustration of the SFG-Tracing Multi-Level Behavioural Verification Methodology, by the Correctness Proof of a High to Low Synthesis Application in Cathedral-II, Proc. IEEE International Conf. on Computer Design: VLSI in Computers and Processors, ICCD'91, Oct. 14-16, 1991, Cambridge, MA.

[9] J. Joyce, Multi-Level Verification of Microprocessor-Based Systems, Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1989. Report No. 195, Computer Laboratory, Cambridge University, May 1990.

[10] M.J.C. Gordon et al., *The HOL System Description*, Cambridge Research Centre, SRI International, Suite 23, Miller's Yard, Cambridge CB2 1RQ, England.

[11] C-J. Seger, "An Introduction to Formal Hardware Verification", Technical Report 92-13, Department of Computer Science, University of British Columbia, June 1992.

[12] C-J. Seger, "The Voss Verification System—User's Guide", in preparation.

[13] C-J. Seger and J. J. Joyce, "A Mathematically Precise Two-Level Formal Verification Methodology", Report 92-34, Department of Computer Science, University of British Columbia, December 1992.

[14] C-J. Seger and R. E. Bryant, "Formal Verification of Digital Circuits by Symbolic Evaluation of Partially-Ordered Trajectories", in preparation.

[15] *Silos II—Logic and Fault Simulator: User's manual*, SIMUCAD, Palo Alto, 1988.