A Mathematically Precise Two-Level Formal Hardware Verification Methodology*

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

Abstract

Theorem-proving and symbolic trajectory evaluation are both described as methods for the *formal verification of hardware*. They are both used to achieve a common goal—correctly designed hardware—and both are intended to be an alternative to conventional methods based on non-exhaustive simulation. However, they have different strengths and weaknesses. The main significance of this paper is the description of a two-level approach to formal hardware verification, where the HOL theorem prover is combined with the Voss verification system. From symbolic trajectory evaluation we inherit a high degree of automation and accurate models of circuit behavior and timing. From interactive theorem-proving we gain access to powerful mathematical tools such as induction and abstraction. The interface between the HOL and Voss is, however, more than just an ad hoc translation of verification results obtained by one tool into input for the other tool. We have developed a "mathematical" interface where the results of the Voss system is embedded in HOL. We have also 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 system in VLSI technology usually involves working at several levels of abstraction, ranging from very high level behavioral specifications down to physical layout at the lowest. 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. Typically the designer tries relatively few test cases, one at a time, and checks whether the results are correct. Towards the end of the design phase, the circuit is then often

*This research was supported by operating grants OGPO 109688 and OGPO O46196 from the Natural Sciences Research Council of Canada, fellowships from the Advanced Systems Institute, and by research contract 92-DJ-295 from the Semiconductor Research Corporation.

simulated for an extended period of time. It is not uncommon to spend months of CPU time on mainframe computers simulating the final design[25]. Despite this tremendous effort, serious design errors often remain undetected. In addition, as designs become more complex through the introduction of aggressive pipelining and concurrently-operating subsystems, it becomes increasingly difficult to anticipate the many very subtle interactions between logically unrelated system activities.

The current situation can be summed up with the observation that the "quality" of the verification achieved by traditional simulation is rapidly deteriorating as the VLSI technology progresses. 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. These methods all have their strengths and weaknesses. In this paper we will illustrate how theorem-proving can be used in conjunction with symbolic simulation to gain a verification methodology that draws on the strengths of each approach.

The paper is organized as follows. We first briefly introduce theorem-proving and symbolic simulation based formal hardware verification. In particular, we focus on the two approaches strengths and weaknesses. We then introduce the underlying theory for our combined approach. We also outline a prototype implementation and describe some of the results we have been able to derive using this prototype. We conclude with a small illustrative example and some conclusions and future directions. Although this paper contains a number of examples, the main focus on our presentation is the "mathematical interface" that we have developed to link theorem-proving with symbolic simulation. For a more comprehensive discussion of how to use the prototype system and some more realistically sized examples, the reader is referred to the companion paper[23].

We believe the development of the mathematical interface between the HOL system and the Voss system can be used as a guide for the development of other hybrid approaches to formal hardware verification. We strongly believe that the development of hybrid tools such as HOL-Voss are essential if formal methods are to ever become more than an academic interest in the VLSI design community.

2 Interactive Theorem Proving

One of the earliest approaches to formal hardware verification was to describe both the implementation as well as the specification in a formal logic. The correctness result was then obtained by using formal rules of reasoning to derive a relationship between the implementation and specification of a design. In this section we will briefly summarize the main characteristics of this form of verification.

Following [24] we define a *formal theory* as follows: A formal theory S is defined when:

1. A finite alphabet is given. The symbols of this alphabet are the *symbols* of the theory. A finite sequence of these symbols is called an *expression* of S.

2. A subset of the expressions of S are called *well-formed formulas* of S.

3. A finite set of the well-formed formulas of S are called *axioms* of S.

4. A finite set of *rules of inference* is given. A rule of inference allows the derivation of a new well-formed formula from a given finite set of well-formed formulas.

A *formal proof* in S is a finite sequence of well-formed formulas: $f_1, f_2, \ldots, f_n$, such that for every i, formula f_i is either an axiom or can be derived by one of the rules of inference given the set of formulas $\{f_1, f_2, \ldots, f_{i-1}\}$. Traditionally, the last well-formed formula in a formal proof is called a *theorem* of S, and the formal proof is a *proof* of this theorem.

To illustrate a formal theory, we will use the formulation of higher-order logic, as used in the HOL theorem prover and described in[18, 19]. The alphabet used contains most of the symbols from predicate logic, i.e., $\vee$, $\wedge$, $\Rightarrow$, $\forall$, $\exists$, etc. However, it also contains symbols for lambda abstraction and various forms of type annotation. Some examples of well-formed formulas are:

$$(9 + 12) : \mathrm{num}$$

$$((9 + 12) = 3) : \mathrm{bool}$$

$$(\lambda x.\ ((x + 12) = 15)) : \mathrm{num} \rightarrow \mathrm{bool}$$

(The last formula denotes a function that takes a number and returns true if and only if the number is 3.) A theorem is written as $\Gamma \vdash t$, and is read as: assumptions Γ imply theorem t.

Like many formal theories, the higher-order logic is defined using a very small number of axioms. In the formulation used in the HOL system, only five primitive axioms are used. These axioms range from, the very simple and "obvious" ones like

$$\emptyset \vdash \forall t{:}\mathrm{bool}.\ (t = T) \vee (t = F)$$

which basically states that a Boolean type is either true or false, to some much less obvious axioms, like

$$\emptyset \vdash \forall P : \star \rightarrow \mathrm{bool}.\ \forall x : \star.\ Px \Rightarrow P(\epsilon P)$$

which states some of the properties of the Hilbert ϵ-operator which basically serves as a "choice" operator[1].

The number of rules of inference is also often quite small. In the formulation of higher-order logic within the HOL system, there are only eight primitive rules of inference. The following three rules are representative examples:[2]

$$\frac{}{t \vdash t}$$

which allows us to make assumptions,

$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash (t_1 \Rightarrow t_2)}$$

which allows us to discharge an assumption, and

$$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

[1]The axiom states essentially that for every predicate P over some domain, if P holds for some element x in the domain, then ϵP denotes a value that satisfies P.

[2]We use the standard convention of writing rules of inference as a horizontal line with the assumptions written above and the conclusion written below.

which formalizes the informal rule of reasoning "if t_1 implies t_2 and t_1 is true, then t_2 must also be true".

It is important to distinguish between proofs in the common mathematical sense and proofs in a formal system. The former often relies on unspoken assumptions, assumed knowledge, and leaps of intuition, whereas every step in the latter is completely justified. Of course, this means that a formal proof is very easy to check. For example, one can imagine writing a very simple proof checker that checks each line of the proof that it is either an axiom or that it has been obtained by a valid rule of inference given the preceding lines. This strength of a formal proof is also its main drawback. Even very simple proofs can take a very large number of steps to carry out. Consequently, trying to derive formal proofs by hand is impractical. However, many of the steps in a formal proof is mostly tedious book-keeping—something computers are very good at—and thus machine assisted theorem provers have been developed.

Among the best known interactive theorem-provers are the Boyer-Moore Theorem Prover [5] and the Cambridge HOL System [18, 19]. The Boyer-Moore Theorem Prover has been used by researchers at Computational Logic Inc. to develop a multi-level proof of correctness for a complete computer system including both hardware and software levels [3]. The Cambridge HOL System has been used by researchers at Cambridge University to verify aspects of the commercially-available Viper microprocessor designed by the British Ministry of Defense for safety-critical applications [13].

One of the main strengths of the theorem-proving approach is its ability to describe and relate circuit behaviors at many different levels of abstraction. For example, when verifying an adder, we can show that the relationship between the inputs and outputs corresponds to addition as defined, for instance, by Peano axioms, rather than just a relationship between Boolean variables. Thus, we can reason at different algebraic levels and relate behaviors between the levels. This point cannot be stressed enough, since one of the main difficulties in formal hardware verification is to convince oneself that the specification is indeed correct. By being able to reason about the circuit at increasingly higher levels of abstraction, we can eventually minimize the semantic gap between the formal high-level specification and the informal, intuitive, specification of the circuit that resides in the mind of the designer.

Unfortunately, theorem-proving based verification requires a large amount of effort on the part of the user in developing specifications of each component and in guiding the theorem prover through all of the lemmas. Also, in order to make the proofs tractable, most attempts at this style of verification have been forced to use highly simplified circuit models.

3 Symbolic Trajectory Evaluation

Symbolic simulation is an offspring of conventional simulation. Like conventional simulation, it uses a built-in model of hardware behavior and a simulation engine to compute, on demand, the behavior of some design for some given inputs. However, it differs in that it considers symbols rather than actual values for the design under simulation. In this way, a symbolic simulator can simulate the response to entire classes of values with a single simulation run.

The concept of symbolic simulation in the context of hardware verification was first proposed by researchers at IBM Yorktown Heights in the late 1970's as a method for evaluating register transfer language representations [12]. The early programs were limited in their analytical power since their symbolic manipulation methods were weak. Consequently, symbolic simulation for hardware verification did not evolve much further until more efficient methods of manipulating symbols emerged. The development of Ordered Binary Decision Diagrams (OBDDs) for representing Boolean functions [9] radically transformed symbolic simulation.

Since a symbolic simulator is based on a traditional logic simulator, it can use the same, quite accurate, electrical and timing models to compute the circuit behavior. For example, a detailed switch-level model, capturing charge sharing and subtle strengths phenomena, and a timing model, capturing bounded delay assumptions, are well within reach. Also—and of great significance—the switch-level circuit used in the simulator can be extracted automatically from the physical layout of the circuit. Hence, the correctness results can link the physical layout with some higher level of specification.

The first "post-OBDD" symbolic simulators were simple extensions of traditional logic simulators [7]. In these symbolic simulators the input values could be Boolean variables rather than only 0's, 1's as in traditional logic simulators. Consequently, the results of the simulation were not single values but rather Boolean functions describing the behavior of the circuit for the set of all possible data represented by the Boolean variables. By representing these Boolean functions as Ordered Binary Decision Diagrams the task of comparing the results computed by the simulator and the expected results became straightforward for many circuits. Using these methods it has become possible to check many (combinational) circuits exhaustively.

Recently, Bryant and Seger [10, 28] developed a new generation of symbolic simulator based verifier. Since the method has departed quite far from traditional simulation, they called the approach *symbolic trajectory evaluation*. Here a modified version of a simulator establishes the validity of formulas expressed in a very limited, but precisely defined, temporal logic. This temporal logic allows the user to express properties of the circuit over *trajectories*: bounded-length sequences of circuit states. The verifier checks the validity of these formulas by a modified form of symbolic simulation.

Although the general theory underlying symbolic trajectory evaluation, as described in [10, 28], is equally applicable to hardware as software systems, we will only describe a somewhat specialized version tailored specifically to hardware verification. For a more comprehensive discussion of the general case, the reader is referred to [10, 28].

In symbolic trajectory evaluation the circuit is modeled as operating over logic levels 0, 1, and a third level X representing an indeterminate or unknown level. These values can be partially ordered by their "information content" as $X \sqsubseteq 0$ and $X \sqsubseteq 1$, i.e., X conveys no information about the node value, while 0 and 1 are fully defined values. The only constraint placed on the circuit model—apart from the obvious requirement that it accurately model the physical system—is monotonicity over the information ordering. Intuitively, changing an input from X to a binary value (i.e., 0 or 1) must not cause an observed node to change from a binary value to X or to the opposite binary value. In extending to symbolic simulation, the circuit nodes can take on arbitrary ternary functions over a set of Boolean variables $\mathcal{V}$. Symbolic circuit evaluation can be thought of as computing circuit behavior for many different operating conditions simultaneously, with each possible assignment of 0 or 1 to the variables in $\mathcal{V}$ indicating a different condition.

Properties of the system are expressed in a restricted form of temporal logic having just enough expressive power to describe both circuit timing and state transition properties, but remaining simple enough to be checked by an extension of symbolic simulation. The basic decision algorithm checks only one basic form, the *assertion*, in the form of an implication $[A \implies C]$; the *antecedent* A gives the stimulus and current state, and the *consequent* C gives the desired response and state transition. System states and stimuli are given as trajectories over fixed length sequences of states.

Each of these trajectories are described with a temporal formula. The temporal logic used here, however, is very limited. A formula in this logic is either:

1. UNC (unconstrained),
2. (a) `n`$=1$ (a node is equal to 1),
   (b) `n`$=0$ (a node is equal to 0),
3. $F_1 \wedge F_2$ (F_1 and F_2 must both hold),
4. $B \rightarrow F$ (the property represented by formula F need only hold for those assignments satisfying Boolean expression B),
5. $\mathbf{N}F$ (F must hold in the next state).

The temporal logic supported by the evaluator is far weaker than that of more traditional model checkers. It lacks such basic forms as disjunction and negation, along with temporal operators expressing properties of unbounded state sequences. The logic was designed as a compromise between expressive power and ease of evaluation. It is powerful enough to express the timing and state transition behavior of circuits, while allowing assertions to be verified by an extended form of symbolic simulation.

The constraints placed on assertions make it possible to verify an assertion by a single evaluation of the circuit over a number of circuit states determined by the deepest nesting of the next-time operators. In essence, the circuit is simulated over the unique weakest (in information content) trajectory allowed by the antecedent, while checking that the resulting behavior satisfies the consequent. In this process a Boolean function is computed expressing those assignments for which the assertion holds.

The assertion syntax outline above is very primitive. To facilitate generating more abstract notations, the specification language can be embedded in a general purpose programming language. When a program in this language is executed, it automatically can generate the low-level temporal logic formulas and carry out the verification process.

The Voss system is a formal verification system based on symbolic trajectory evaluation developed at University of British Columbia. Conceptually, the Voss system consists of two parts as shown in Figure 1. The front-end is a compiler/interpreter for a small, fully lazy, functional language. A specification is written as a "program" in this language. When this specification program is executed, i.e., reduced to normal form, it builds up the simulation sequence that must be run in order to completely verify the specification.

The back-end of the Voss system is an extended symbolic simulator. The simulator uses an externally generated finite state machine description to compute the needed trajectories. This finite-state machine is a behavioral model of a digital circuit which can be automatically generated from a transistor netlist by a separate tool called Anamos [8] or from a gate netlist in Silos format[29] by a program called silos2exe. Since we are using the Anamos tool to precompile a switch-level netlist, the Voss system can carry out switch-level verification using

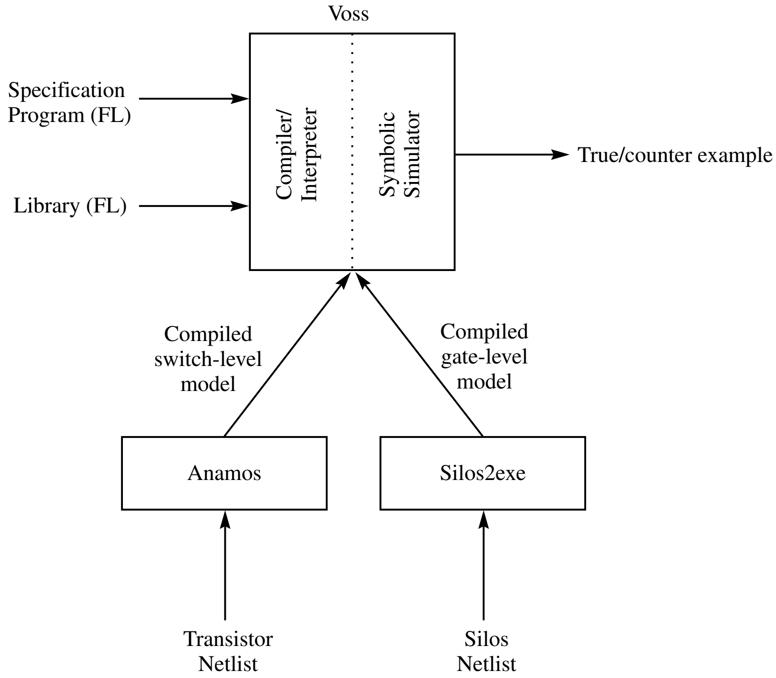

Figure 1: Voss verification system

the full MOSSIMII [6] switch-level model. The gate level simulator is (roughly) functionally equivalent to the SILOS II [29] simulator. In addition, fairly comprehensive delay modeling capabilities has been added for more accurate verification. In order to achieve good performance, the symbolic simulator employs event scheduling for both the circuit simulation as well as in maintaining the verification conditions

From the Voss user's point of view the Voss system verifies assertions of the form,

```
FSM fsm (antecedent,consequent)
```

where `fsm` denotes a finite-state machine and the pair `(antecedent,consequent)` expresses a relationship over the trajectories of this finite-state machine. The parameters `antecedent` and `consequent` each denotes a list of "atomic" constraints. 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 for one particular input value and where the output value is delayed by 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`.

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. 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`".

Symbolic trajectory evaluation does not require a hierarchy of behavioral specifications to be formulated to match the structural hierarchy of the design. In fact, it is often sufficient to use a specification expressed in terms of the behavior of the complete circuit in response to different inputs and starting states. Consequently, the same specification and verification program can often be used for quite different implementations. Also, the exact details of how the circuit works can often be left to the simulator to compute. This makes it possible to *use* sophisticated circuit models without having to know the details of the model. In summary, symbolic trajectory evaluation is a highly automated verification methodology.

Unfortunately, this automation comes with a price. First of all, for some behaviors, the computational requirements for carrying out a correctness proof can make the approach infeasible for larger circuits. For example, verifying a circuit implementing 32-bit integer multiplication is impossible using current symbolic trajectory evaluators. The reason is simply that the Ordered Binary Decision Diagrams used to represent the outputs of the multiplier grows exponentially in the size of the multiplicands, and thus is completely out of reach for a 32 bit version.

A second serious drawback with symbolic trajectory evaluation is that the semantic gap between the intuitive, informal, specification the designer has in mind and the specification used in the symbolic trajectory evaluator is often undesirably large.

4 Pros and Cons of HOL and Voss

In this section we will summarize the various pros and cons with using interactive theorem proving and symbolic trajectory evaluation for formal hardware verification. To make the discussion more concrete, we will use HOL and Voss as examples of tools used for the two approaches.

In comparing the theorem proving approach with symbolic trajectory evaluation we can conclude the following:

- In both cases, verification is a matter of relating a bottom-level specification of an implementation to a top-level specification of its intended function; these specifications are mathematical models.

- In both cases, the users supplies both the bottom and top levels of specification.

- In symbolic simulation, expressiveness, with respect to the representation of data, is generally fixed at the level of Boolean expressions.
- In theorem-proving, specification languages are typically very expressive; theorem-proving methods generally allow different representations of data to be formally related at increasing levels of abstraction.
- In both cases, verification is governed by a fixed set of symbol manipulation rules.
- In symbolic simulation, verification is completely automatic; a principal concern is efficiency.
- In theorem-proving, verification is interactive and usually depends on the user to guide the theorem-prover through a high-level proof strategy; a principal concern is user control of the verification process.
- In symbolic simulation, verification is not tightly coupled to the hierarchical structure of a design.
- In theorem-proving, verification is tightly coupled to the hierarchical structure of a design.
- In symbolic simulation, complexity is controlled by the use of an efficient internal representation with a canonical form.
- In theorem-proving, complexity is controlled by a variety of mechanisms including induction and hierarchical proof strategies.
- In symbolic simulation, performance of the verification process is determined mainly by the speed and size of the host machine.
- In theorem-proving, performance is determined mainly by the expertise of the user.

Based the above points, one may conclude that symbolic simulation is a highly restricted, but very efficient method of formally verifying hardware while theorem-proving is a very flexible, but less automatic and less efficient method.

In particular, we regard the ability to reason about data at increasing levels of abstraction to be a major strength of theorem-proving. On the other hand, attempts to reason about detailed level circuit behavior are generally very difficult for theorem-proving methods—and the results are not very convincing.

The strengths and weaknesses of symbolic simulation are exactly the opposite: symbolic simulation does not support abstraction representations of data but it can be used to reason about detailed circuit level behavior very efficiently—and the the results are indeed convincing. Hence, an ideal verification tool would draw from both methodologies. In the remaining parts of this paper we will indeed describe exactly such a hybrid approach to formal hardware verification.

5 Linking HOL to Voss

Our efforts in developing a two-level verification methodology focused initially on the establishment of a mathematical link between the underlying logical framework of the HOL system and the notion of symbolic trajectory evaluation, as implemented in the Voss system. This mathematical link was then used as the basis for the implementation of a prototype verification tool that extends the built-in functionality of the HOL system with additional facilities for reasoning about very large finite-state machines using symbolic trajectory evaluation. From a functional point of view, the HOL-Voss system can be described in two ways. First, one can view the HOL-Voss system as a specialized extension of the HOL system. On the other hand, one can also view the HOL system as a tool that can be used to develop, provably correct, verification procedures to be used in the Voss system. In this paper we will primarily take the first view, but we will return to the second view in Section 8. The current HOL-Voss system is actually implemented as the integration of HOL with the Voss system where the HOL system delegates certain proof tasks to a concurrently executing Voss system through a process-to-process communication channel.

5.1 Semantic Embedding of Voss in HOL

The cornerstone of our hybrid approach is the establishment of a mathematical link between higher-order logic and the notion of symbolic trajectory evaluation. This link is established by defining several new types (`noderef`, `nodevalue`, `state`, `trajectory`, and `fsm`) and several new predicates (`Trajectory`, `FSM` and `SuffixClosed`). In HOL jargon, this link can be described as a "semantic embedding" of Voss within higher-order logic.

A finite-state machine is represented in our approach by a set of trajectories. A trajectory is an infinite sequence of states where a state is a mapping from node names to node values. A set of trajectories represents a finite-state machine in the sense that every trajectory in the set is a possible sequence of states that might be observed for the finite-state machine. Also, any possible sequence of states that might be observed for this machine exactly matches one of the trajectories in the set.

Table 1: Basic types for the Voss semantics.

```
noderef    = :string
nodevalue  = :bool
state      = :string ->nodevalue
trajectory = :num ->state
fsm        = :trajectory ->bool
```

This representation of a finite-state machine is formalized in the HOL logic with a collection of HOL types as shown in Table 1. Node names are represented by `:string`, a built-in HOL type. Similarly, node values are represented by `:bool`, also a built-in HOL type. The function type, `string`→ `nodevalue` is used to represent states. A trajectory is represented by the function type `num`→ `state`, which represent functions that map natural numbers to states; in this representation, natural numbers are used to represent the position of the state in the state sequence. A finite-state machine, is represented in the HOL logic by the characteristic function for a set of trajectories[3]: `trajectory→ bool`.

The next step in the development of our semantic embedding of Voss within higher-order logic is the definition of a predicate called `Trajectory`. For a given trajectory `tj` and a given 5-tuple `(b,n,v,s,f)`, this predicate expresses the condition that "if the Boolean expression `b` is true, then the node named by `n` has the value `v` in all states of the trajectory `tj` from the start state `s` up to but not including the final state `f`". More formally,

```
⊢def ∀tj b n v s f.
      Trajectory tj(b,n,v,s,f) = b ⇒ (∀i. s ≤ i ∧ i < f ⇒ (tj i n = v))
```

The principal operator for expressing assertions about finite-state machines in our hybrid approach is the predicate `FSM`. For a given finite-state machine, `fsm`, this predicate expresses the condition that "every trajectory of this finite-state machine that satisfies a set of constraints represented by a list of 5-tuples called the `antecedent` also satisfies a set of constraints represented by a list of 5-tuples called the `consequent`". Formally, `FSM` is defined as:

```
⊢def ∀fsm antecedent consequent.
       FSM fsm (antecedent,consequent) =
         let p (tj,l) = ITLIST (λa b. a ∧ b) (MAP (Trajectory tj) l) T in
         ∀tj. fsm tj ∧ p (tj,antecedent) ⇒ p (tj,consequent)
```

The above definition of `FSM` requires some explanation since it involves the use of two pre-defined functions, `MAP` and `ITLIST`, which may not necessarily be familiar to the reader. The sub-expression, `MAP (Trajectory tj) l` denotes the application of the `Trajectory` predicate for a given trajectory, `tj`, to a set of constraints represented by a list of 5-tuples, `l`. This sub-expression evaluates to a list of Booleans values. We then iteratively combine this list of Boolean values by logical conjunction using the higher-order function `ITLIST`:

```
ITLIST (λa b. a ∧ b) (MAP (Trajectory tj) l) T
```

This sub-expression will evaluate to `T`, that is, "true", if and only if every constraint in the set represented by the list `l` is satisfied by the trajectory `tj`. A let-expression is used in the definition of `FSM` to allow this sub-expression to be used for both the `antecedent` and the `consequent`.

The final component of our semantic embedding of Voss within higher-order logic is the definition of the predicate `SuffixClosed`:

```
⊢def ∀fsm. SuffixClosed fsm = (∀tj. fsm tj ⇒ fsm(λn. tj(n+1)))
```

This predicate expresses the condition that the suffix of every trajectory of the finite-state machine—that is, the result of chopping off any finite initial sub-sequence of the trajectory—is also a trajectory of the finite-state machine. This condition is equivalent to the condition that every possible state of the finite-state machine is a possible initial state.

The semantic embedding of Voss within higher-order logic causes the specification language of Voss to become a subset of the specification language of the HOL system. For example, the assertion,

[3]A set of trajectories can also be used to model state machines with infinite state, however we only intend to ever use this representation for finite-state machines.

```
∀v. FSM inverter ([(T,‘input‘,v,0,1)],[(T,‘output‘,¬v,1,2)])
```

becomes a term of higher-order logic as a result of this semantic embedding. For a given value of `inverter`, that is, the specification of the constant `inverter` as a finite-state machine, the above assertion is either true or false. If the assertion is true, then it is a theorem of higher-order logic. The goal of our hybrid approach—at a very fundamental level—is to extend the HOL system with an additional proof procedure based on symbolic trajectory evaluation for verifying assertions about finite-state machines expressed in terms of `FSM`. This proof procedure, called `VOSS_TAC`, can be used to verify that such assertions are theorems of higher-order logic.

5.2 A Functional View of HOL-Voss

From a functional point of view, the HOL-Voss system can be described in terms of two fundamental extensions to the HOL system:

- a special-purpose specification mechanism, `new_fsm_specification`, for creating specifications of finite-state machines,
- a special-purpose proof procedure, `VOSS_TAC`, for verifying assertions about finite-state machines expressed in terms of the predicate `FSM`.

There is tight coupling between these two fundamental extensions. The proof procedure `VOSS_TAC` can be used to verify assertions about a finite-state machine only when the specification of this finite-state machine has been created using `new_fsm_specification`. Furthermore, no proof procedure except `VOSS_TAC` has access to specifications created by means of `new_fsm_specification`.

The HOL system provides several standard mechanisms for creating specifications. These specification mechanisms are used to introduce new constants and partially specify the value of these constants. For example, the HOL system provides a function, `new_definition`, that could be used to introduce a new constant `TWO` and specify that this new constant is equal to the value of the built-in constant `2`. Evaluation of the meta-expression,

```
    new_definition (‘TWO‘,"TWO = 2");;
```

in a HOL session would cause an internal database to be extended with both a new constant, `TWO`, and a new axiom, ⊢ `TWO = 2`.

In a similar way, the special-purpose specification mechanism provided by HOL-Voss can be used to introduce a new constant and give a partial specification of its value. For example, evaluation of the meta-expression,

```
new_fsm_specification (‘inverter‘);;
```

in a HOL-Voss session has the effect of introducing a new constant, `inverter`, and specifying the value of this constant as an instance of a finite-state machine. However, unlike standard HOL specification mechanisms where the specification of this constant would be given by an explicit term of higher-order logic, HOL-Voss looks for a specification of this finite-state machine in an external file called `inverter.exe`. The expected format of this external file does not involve an explicit term of higher-order logic; instead, this file is an encoded representation of a set of next-state equations in a format produced by one of the circuit compilers in the Voss system.

From a functional point of view, the result of using `new_fsm_specification` is similar to the result of using a standard HOL specification mechanism such as `new_definition`. After evaluating the above expression to create a specification for the inverter circuit, the HOL-Voss user could inspect the database and see that a new constant, `inverter`, has been introduced. Furthermore, the HOL-Voss user would see that this constant has been partially specified by the introduction of a new axiom,

```
⊢ SuffixClosed inverter
```

which specifies that every state of the finite-state machine model of the inverter is a possible initial state of the machine. However, the rest of the finite-state machine specification, in particular, that part of the specification which specifies the actual behavior of the machine, is stored in a second database maintained by HOL-Voss exclusively for the purpose of storing specifications created by means of `new_fsm_specification`. The first database corresponds to the database of a standard HOL system; its contents have a printable representation and can be accessed by all of the facilities inherited by HOL-Voss from HOL for doing interactive proofs in higher-order logic. However, the specifications in the second database do not have a printable representation as terms of higher-order logic. Furthermore, these specifications can only be accessed by the special-purpose proof procedure, `VOSS_TAC`.

The proof procedure `VOSS_TAC` is a HOL-Voss extension of the HOL system that can be used to automatically verify assertions about finite-state machines specified by means of `new_fsm_specification`. For example, if our current proof goal is of the form:

```
∀v. FSM inverter ([(T,‘input‘,v,0,1)],[(T,‘output‘,¬v,1,2)])
```

evaluation of the meta-expression,

```
e (VOSS_TAC ‘inverter‘);;
```

would cause HOL-Voss to use symbolic trajectory evaluation to check the validity of this assertion. If this assertion is found to be true, then evaluation of this meta-expression would result in the generation of a new theorem,

```
⊢ ∀v. FSM inverter ([(T,‘input‘,v,0,1)],[(T,‘output‘,¬v,1,2)])
```

with a status equal to theorems generated using only standard HOL proof procedures.

In the above example, the `antecedent` and `consequent` each consist of just one 5-tuple. However, HOL-Voss is intended to be used to verify assertions about very complex finite-state machines where the `antecedent` and `consequent` may each consist of thousands of 5-tuples. For this reason, notational conveniences built-in the HOL logic such as as let-expressions may be used in assertions submitted to `VOSS_TAC`. This is illustrated, for example, in the following assertion about the implementation of a full-adder:

```
∀a b c.
    let xor (n,m) = (n ∧ ¬m) ∨ (¬n ∧ m) in
    let sum (a,b,c) = xor (xor (a,b),c) in
    let cout (a,b,c) = (a ∧ b) ∨ (a ∧ c) ∨ (b ∧ c) in
    let antecedent = [(T,‘a‘,a,0,20);
                      (T,‘b‘,b,0,20);
                      (T,‘cin‘,c,0,20)] in
    let consequent = [(T,‘sum‘,sum (a,b,c),10,20);
                     (T,‘cout‘,cout (a,b,c),10,20)] in
    FSM fulladder (antecedent,consequent)
```

Prior to verifying an assertion by means of symbolic trajectory evaluation, `VOSS_TAC` will automatically apply a number of general-purpose reduction rules. Moreover, `VOSS_TAC` will use the definitions of user-defined constants to reduce an assertion prior to checking the validity of this assertion by means of symbolic trajectory evaluation. The mechanism for these reductions is explained in the next section.

5.3 An Implementation View of HOL-Voss

The HOL-Voss system is implemented as the integration of HOL with the Voss system where the HOL system delegates certain proof tasks to a concurrently executing Voss system through a process-to-process communication channel. In our Unix-based implementation, the HOL part of our implementation invokes Voss as a child process and communicates with Voss by means of a pseudo-tty interface. The user interacts with HOL-Voss exclusively through the standard HOL interface. Certain user interaction, in particular, the specification of a finite-state machine using `new_fsm_specification` or the application of `VOSS_TAC` to an assertion about a finite-state machine, cause the HOL process to issue commands to Voss using this pseudo-tty interface.

When `new_fsm_specification` is used to create a specification of a finite-state machine, the HOL part of our HOL-Voss implementation commands Voss to read in the specification of the finite-state machine from the external `.exe` file. Earlier when describing the functional view of the HOL-Voss system, it was explained that finite-state machine specifications created using `new_fsm_specification` are stored in a separate database. From an implementation point of view, it can be seen that these finite-state machine specifications are stored as data structures within the Voss process while specifications created using standard HOL specification mechanisms such as `new_definition` are stored separately as data structures in the HOL process.

When `VOSS_TAC` is applied to an assertion about a finite-state machine, the assertion is passed by means of the software interface from the HOL part of our hybrid tool to Voss and Voss is then commanded to verify the assertion using symbolic trajectory evaluation. The assertion is passed from HOL to Voss as a type-checked parse tree for the corresponding HOL term. We have extended Voss to read such HOL terms directly and try to execute them. In addition, every constant in the HOL system introduced by means of a definition is also written out as a (type-checked) parse tree and stored as a `let` (`letrec` for recursive definitions) in a Voss library. In fact, for every HOL theory the user builds up, there will be a Voss library containing the definitions of all constants defined in the theory.

A number of HOL constants are built into the Voss system; these constants are listed in Table 2. For Voss to be able to deal with any new constant, it has to be defined in terms of these known constants. The only major restriction of the constants is that the universal and existential quantifiers must be of type `:(bool→ bool)→ bool`, i.e., quantification can only be done over a Boolean variable. Note that the only "new" constant introduced in HOL-Voss, i.e., a constant that is not part of standard HOL, is `FSM`.

Table 2: Built-in HOL constants recognized by Voss.

CONS	$\forall$	[1-9][0-9]*	$=$
HD	$\exists$	$+$	$>$
TL	COND	$-$	$\geq$
NIL	F	$*$	
	T	DIV	,
STRING	$\neg$	MOD	FST
'...'	$\wedge$	SUC	SND
	$\vee$	$<$	
FSM	$\Longrightarrow$	$\leq$	

When the Voss system receives a HOL parse tree for a goal to verify, it performs two actions. First it reads in all the Voss libraries for the HOL theories used. It then tries to reduce the obtained "functional program" to normal form. In the process, it will expand all user-defined constants into the built-in constants, and execute the primitive operations. For example, evaluation of the HOL meta-expressions

```
let z = new_definition(`z`, "z = 0");;
let factorial = new_prim_rec_definition(`factorial`,
                 "(factorial 0 = 1) ∧
                  (factorial (SUC n) = (SUC n) * (factorial n))");;
```

would, in addition to storing the definitions in the HOL database, cause the definitions[4]

```
let z = 0 in
letrec (factorial 0 = 1) ∧ (factorial (SUC n) = (SUC n) * (factorial n)) in
```

to be stored in a Voss library file. Next, if the HOL-Voss user tries to prove the goal `"(factorial 5) + z = 120"` by giving the meta-expression

```
e (VOSS_TAC ``);;
```

the following will happen. First, Voss will read in all the Voss libraries that contain user-defined constants. In particular, Voss will read in the definitions of `z` and `factorial` shown above. HOL will then send Voss the parse tree for the term `(factorial 5) + z = 120`. At this point, Voss will be faced with the task of evaluating:

```
let z = 0 in
letrec (factorial 0 = 1) ∧ (factorial (SUC n) = (SUC n) * (factorial n)) in
(factorial 5) + z = 120
```

[4]For readability, we have rewritten the parse trees in infix notation.

Voss will do so by computing that `(factorial 5)` evaluates to 120, `z` evaluates to 0, `(factorial 5) + z` evaluates to 120, and finally that `(factorial 5) + z = 120` evaluates to `T`. Note that in this example we did not use any of the symbolic trajectory evaluation facilities of Voss. Normally the goal will of course contain at least one instance of `FSM` causing Voss to carry out a complete symbolic trajectory evaluation.

There are some restrictions on what kind of definitional specification that can be executed by Voss. These restrictions come in three forms:

- quantification can only be done over Booleans (the same holds for the Hilbert ϵ-operator),
- when a `COND` (conditional) is evaluated, the condition cannot contain any free variables, and
- currently Voss is using fixed precision arithmetic and thus some provably correct goals cannot be proven.

The last restriction is annoying but fairly straightforward to remove. In fact we are currently considering replacing the fixed precision arithmetic in the Voss system with arbitrary precision arithmetic. Once this is done, this restriction can be lifted. Note however, that the current version checks for overflows and thus there is no danger of "proving" incorrect results due to overflows.

The second limitation is a consequence of the fact that we are not able to perform symbolic evaluation of HOL terms. Thus, a term like:

```
(a ∨ b) →  ¬c | ¬b
```

cannot be evaluated since both `a` and `b` are free in the condition (`a`∨`b`). Fortunately, it is often fairly straightforward to replace these kinds of constructs with other Boolean operations. For example, one can prove the following theorem

```
⊢ ((a ∨ b) →  ¬c | ¬b) = (a ∨ b ⇒ ¬c) ∧ (¬(a ∨ b) ⇒ ¬b)
```

and using this result, we can rewrite the original goal to a form acceptable to Voss. Of course, it is not always possible to find an equivalent form acceptable to Voss, and this limitation remains as one of the most difficult to deal with.

Finally, as already mentioned, the Voss system can only perform quantification over Booleans. Hence, all other quantifications must be rephrased in terms of quantifications over Booleans. We will return to this later in this paper.

If the goal is verified by Voss, it returns the result `T`, that is, "true", back to the HOL system through the software interface; otherwise, the result `F` is returned to indicate that Voss was unable to verify the goal. If the goal involves symbolic trajectory evaluation, the Voss system will also print out debugging information and, most importantly, if the verification fails, a counterexample is given of the form of a node that does not have the right value at the right time.

5.4 Implementation Tradeoffs

The semantic embedding of the Voss specification language in higher-order logic is just one of many examples of special-purpose formalisms that have been semantically embedded in the logical framework of the HOL system. One of several published examples is the semantic embedding of CSP, a process algebra widely used for formal reasoning in the areas of concurrency, communication, and distributed systems, in the HOL system [11]. The first step in the development of a semantic embedding is to define a basic set of operators as predicates or functions in the underlying framework of higher-order logic. Next, proof procedures of the embedded formalism are implemented as proof procedures in the HOL system. As illustrated in the CSP example, the conventional implementation strategy for extending the HOL system with a new proof procedures is to introduce a new HOL proof procedure as a procedural abstraction defined ultimately in terms of built-in proof procedures of the HOL system. The main advantage of this conventional implementation strategy is the ease of arguing that the introduction of a new proof procedure in this manner does not compromise the integrity of the HOL system: if the new proof procedure simply calls a sequence of built-in HOL proof procedures, then it is easy to see that this new proof procedure cannot be used to derive any theorem that is not already derivable from built-in HOL proof procedures.

In the case of HOL-Voss, we have extended the HOL system with a single proof procedure, `VOSS_TAC`, for verifying assertions about finite-state machines. In principle, this new proof rule could be implemented using the conventional implementation strategy outlined above—that is, as a procedural abstraction defined ultimately in terms of built-in proof procedures. However, we decided against this implementation strategy for several reasons. Most importantly, we decided that it would be an overwhelming disadvantage to re-implement the symbolic trajectory algorithm rather than making direct use of the existing state-of-the-art implementation provided by the Voss system. Also, creating the equivalent to `VOSS_TAC` based solely on the built-in proof procedures would be prohibitively slow, since ultimately, all the proofs would have to be constructed from very elementary inference rules. For example, even in verifying relatively modest circuits, the Voss system often performs hundreds of thousands or even millions of Boolean function applications and comparisons. Replacing these, highly optimized, BDD manipulations with primitive rules of inference, is possible in theory, but is clearly not a practical solution.

There is a more important reason why implementing a symbolic trajectory evaluator in the HOL system may not be the right approach. The current implementation of HOL-Voss appears to the user as a theorem prover on which a symbolic trajectory evaluator has been crafted in. However, there is another way of looking at our general approach. Consider using the HOL system as an "expert tool" used to derive libraries and proof procedures to be used by the Voss system. Here, the main tool for day-to-day verification is the Voss system. Hence, the Voss system with its much simpler user-interface and high degree of automation is the verification tool used by the designers, whereas the HOL prover is used by a smaller group of theorem-proving experts to develop suitable proof infrastructure and provably correct proof procedures to be used in the Voss system.

6 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 pre-defined bitvector arithmetic functions and proofs,
- HCL—a small (experimental) specification language, and
- general proof procedures for common verification tasks in the HOL-Voss system.

In this section we will expand upon these topics.

6.1 Bitvector Arithmetic

Many, if not most, circuits perform some arithmetic on bitvectors. In these cases, it is natural to use arithmetic functions on bitvectors in the specification of the desired circuit operation. In order to increase re-usability and simplify the task of writing specifications, we have developed a "library" of bitvector operations. We represent bitvectors as lists of Booleans and use a "little-endian" view of the bitvectors, i.e., the least significant bit is the head of the list. We have so far limited ourselves to define relational and arithmetic operations for unsigned bitvectors. In Table 3 we list some of the functions from the little-endian library[5]. Note that we not only define relational and arithmetic functions, we also define functions that translate natural numbers, as defined in the HOL system, back and forth to the bitvector representation.

Since these arithmetic bitvector functions are likely to be used over and over again in writing specifications, it is paramount that their definitions are correct. In most model checking or symbolic simulation based verification systems, we would have to ensure this correctness manually. In fact, it is fairly unlikely that we would actually carry out a formal correctness proof. In our case, however, we can formally prove that they correspond to our standard notion of addition, multiplication, etc. as defined by Peano's axioms. In Table 4 we list a collection of the theorems that we have proven about our definitions. For example, the theorem `BV2NUM_NUM2BV` states that for any natural number n, if we first convert n to our bitvector representation using the `num2bv` function and then apply the function `bv2num` to the resulting bitvector, we get n back. Similarly, the theorem `BVPLUS_OF_BVNUMS` states that, for any two bitvectors a and b, the result of applying `bvplus` to these bitvectors and converting the result to a natural number using the `bv2num` function yields the same number as converting the bitvectors a and b to numbers and then adding them together.

Using the above correctness theorems allow us to develop some general proof procedures. In particular, we have develop a procedure called `BV_ARITH_TAC` that can rewrite a goal stated in terms of arithmetic functions and relations to a goal in terms of bitvector versions of the operations. In Section 7 we illustrate the use of `BV_ARITH_TAC`.

It should be noted that the above specifications are in fact the bitvector specifications directly used by the Voss system. Since the Voss system reads (type checked) HOL definitions

[5]We have also developed a "big-endian" library as a trivial extension of the little-endian library.

Table 3: Subset of little endian bitvector operations.

```
⊢def (bv2num[] = 0) ∧ (∀h t. bv2num(CONS h t) = (h →  1 | 0) + (2 * (bv2num t)))

⊢def (∀m. num2bv_aux 0 m = []) ∧
      (∀n m. num2bv_aux(SUC n)m =
        ((m = 0) →  [] | CONS(m MOD 2 = 1)(num2bv_aux n(m DIV 2))))

⊢def ∀n. num2bv n = num2bv_aux n n

⊢def (∀c. bvplus2_aux[]c = [c]) ∧
      (∀h r c. bvplus2_aux(CONS h r)c = CONS(c ∧ ¬h ∨ ¬c ∧ h)(bvplus2_aux r(c ∧ h)))

⊢def (∀b c. bvplus_aux[]b c = bvplus2_aux b c) ∧
      (∀h r b c. bvplus_aux(CONS h r)b c =
        ((b = []) →  bvplus2_aux(CONS h r)c |
          CONS (h ∧ ¬HD b ∧ ¬c ∨ ¬h ∧ HD b ∧ ¬c ∨ ¬h ∧ ¬HD b ∧ c ∨ h ∧ HD b ∧ c)
            (bvplus_aux r(TL b)(h ∧ HD b ∨ c ∧ HD b ∨ h ∧ c))))

⊢def ∀a b. a bvplus b = bvplus_aux a b F

⊢def (∀av. av bvmult [] = [F]) ∧
      (∀av h r. av bvmult (CONS h r) =
        (MAP(λv. h ∧ v)av) bvplus (CONS F(av bvmult r)))

⊢def (∀b res. bvgreater_aux[]b res = res ∧ bvequal_zero b) ∧
      (∀h t b res. bvgreater_aux(CONS h t)b res =
        (NULL b →  (res ∨ h ∨ ¬bvequal_zero t) |
           bvgreater_aux t(TL b)(h ∧ ¬HD b ∨ res ∧ (h = HD b))))

⊢def ∀a b. a bvgreater b = bvgreater_aux a b F

⊢def ∀a b. a bvgeq b = bvgreater_aux a b T
```

Table 4: Sample of correctness theorems for bitvector operations.

```
BV2NUM_NUM2BV =
    ⊢ ∀n. bv2num(num2bv n) = n
SIZED_NUM2BV_BV2NUM_IS_SIZED =
    ⊢ ∀n b. sized n(num2bv(bv2num b)) = sized n b
BVPLUS_OF_BVNUMS =
    ⊢ ∀a b. (bv2num a) + (bv2num b) = bv2num(a bvplus b)
BVTIMES_THM  =
    ⊢ ∀a b. (bv2num a) * (bv2num b) = bv2num(a bvmult b)
BVGREATER_THM  =
    ⊢ ∀a b. (bv2num a) > (bv2num b) = a bvgreater b
BVGEQ_THM  =
    ⊢ ∀a b. (bv2num a) ≥ (bv2num b) = a bvgeq b
```

directly, without further translation, the link between HOL arithmetic and the bitvector definitions used in the Voss system is very strong.

An interesting side effect of our work on the combined HOL-Voss system, is libraries of pre-proven bitvector definitions. These libraries can equally well be used when using the Voss system or the HOL system as stand-alone verification tools. Hence, the effort required to establish the correctness results described above, can be amortized over many different application areas very quickly. Also, since these correctness proofs are only done once, the effort needed is clearly worth it for the added confidence we gain.

There is one more reason for carrying out the above correctness proofs. It allows us to write our high-level specifications in terms of arithmetic relations and operations and then, using the pre-proven theorems, rewrite the proof obligations to bitvector versions, which eventually will be used in the Voss system for carrying out the verification task. This is important in order to minimize the semantic gap between the high level formal specification and the intuitive specification residing in the head of the designer. We will return to this later.

6.2 HCL

We saw earlier how Voss specifications can be written explicitly in terms of 5-tuple lists. However, writing specifications in this manner is too cumbersome for large complex specifications. Instead, we envision the development of higher-level specification languages which will be automatically compiled into Voss specifications. We anticipate that some of these higher-level specification languages will be subsets of conventional hardware description languages while others might be more experimental specification languages. The only requirement is that there must be an algorithm for compiling the higher-level specifications into lists of 5-tuples in a manner that preserves the meaning of the higher-level specification.

To ensure the integrity of this approach, the higher-level specification language must be semantically embedded in HOL logic. The semantic embedding of this language will associate a semantic function with every constructor of the language. Additionally, a compiler for the higher-level specification language will be implemented by a set of functions defined in higher-order logic; these functions will compile higher-level specifications into Voss specifications expressed in terms of 5-tuple lists. The integrity of the compiler functions can be ensured by proving a "compiler correctness" result to show that the result of compiling a higher-level specification into a list of 5-tuples corresponds to the meaning of the higher-level specification given by the semantic functions.

The semantic embedding of higher-level specification languages and use of compiler correctness techniques to ensure integrity is an adaptation of work described in [14, 20, 21, 22] on compiler correctness techniques for higher-level programming languages. We hope that many, if not most, of the higher-level specification languages for HOL-Voss will be developed by HOL-Voss users rather than ourselves. Of course, this work will generally be done by theorem-proving experts responsible for building infrastructure rather than regular users responsible for using HOL-Voss to verify circuits.

To illustrate how a higher-level specification language can be built on top of HOL-Voss in this manner, we have developed the example of very simple – but very useful — higher-level specification language called HCL (for Higher-level Constraint Language).

HCL provides a number of constructors to improve the readability of specifications. For example, the constructor `is` is used in the HCL expression,

```
    ('in' is T) during (3,10)
```

to directly express the constraint that the value of node 'in' is `T` from time `3` to time `10`. HCL also provides support for the concise specification of constraints about vectors of nodes. For example, instead of writing out thirty-two separate 5-tuples to associate a vector of node values $\mathtt{X} = [x_0, x_1, \ldots, x_{31}]$ with a vector of nodes, `RegA.0`, `RegA.1`, ..., `RegA.31`, the HCL constructor `is_vec` can be used to express this constraint: `(nodevec 32 'RegA') is_vec X`. The HCL constructors `is` and `is_vec` are both used to specify instantaneous constraints. The HCL constructors `during`, along with `for`, `when`, `andc` and `then`, are used to specify temporal constraints, that is, the application of instantaneous constraints over durations of time. For example, the constructors `is_vec`, `during` and `when` could be used in the HCL expression,

```
    (((nodevec 32 'RegA') is_vec X) during (33,78)) when B
```

to express the constraint that the contents of `RegA` are equal to `X` from time 33 until time 78 only if the Boolean variable `B` is true.

In Table 5 we give an abbreviated BNF for the abstract syntax of HCL.

Table 5: Abbreviated BNF for the HCL language.

```
hcl_inst ::= UNC |
             node is bool_expr |
             node_list is_vec bool_expr_list

hcl_stmt ::= hcl_inst for num |
             hcl_inst during (num,num) |
             hcl_stmt then hcl_stmt |
             hcl_stmt when bool_expr |
             hcl_stmt andc hcl_stmt
```

Using standard HOL techniques for embedding programming language-like notations in HOL [14, 20], we have formalized the abbreviated BNF given in Table 5 by defining two recursive data types: the data type `hcl_inst` is used to represent the parse trees for instantaneous constraints while the data type `hcl_stmt` is used to represent the parse trees for temporal constraints. The actual definitions of these types are given in Table 6.

While the constructors `UNC`, `IS`, `IS_VEC`, `FOR`, `DURING`, `WHEN`, `ANDc` and `THEN` could be used directly to write down HCL specifications, we have defined aliases for these constructors that, in some cases, provide infix versions of these constructors. The definitions of these aliases are given in Table 7.

Given the definitions above, the HOL parser would now be able to parse HCL programs and build up a parse tree. The next step in our development of HCL was to define a pair of semantic functions which associate meanings with HCL parse trees. The semantic function for HCL instantaneous statements, `Sem_HCL_INST`, takes a parse tree and returns a

Table 6: Recursive types for HCL.

```
let hcl_inst_Axiom = define_type `hcl_inst_Axiom`
        `hcl_inst =
          UNC |
          IS noderef nodevalue |
          IS_VEC (noderef)list (nodevalue)list`;;

let hcl_stmt_Axiom = define_type `hcl_stmt_Axiom`
        `hcl_stmt =
          FOR num hcl_inst |
          DURING num num hcl_inst |
          WHEN hcl_stmt bool |
          ANDc hcl_stmt hcl_stmt |
          THEN hcl_stmt hcl_stmt`;;
```

Table 7: Constructor functions for HCL.

```
unc    ⊢def unc = UNC
is     ⊢def ∀n v. n is v = IS n v
is_vec ⊢def ∀nl vl. nl is_vec vl = IS_VEC nl vl
for    ⊢def ∀n ht. for n ht = FOR n ht
during ⊢def ∀ht n m. ht during (n,m) = DURING n m ht
when   ⊢def ∀l b. l when b = WHEN l b
andc   ⊢def ∀l1 l2. l1 andc l2 = ANDc l1 l2
then   ⊢def ∀l1 l2. l1 then l2 = THEN l1 l2
```

predicate over states. This predicate determines whether the state satisfies the instantaneous HCL specification represented by the parse tree. The semantic function for HCL temporal constraints, `Sem_HCL`, takes a parse tree and returns a predicate over trajectories. This predicate determines whether the trajectory satisfies the temporal constraint represented by the parse tree.

```
⊢def
   (         Sem_HCL_INST UNC           = unc_sem) ∧
   (∀s v.    Sem_HCL_INST(IS s v)       = is_sem s v) ∧
   (∀nl vl. Sem_HCL_INST(IS_VEC nl vl) = is_vec_sem nl vl)

⊢def
   (∀n ht.  Sem_HCL(FOR n ht)       = for_sem n(Sem_HCL_INST ht)) ∧
   (∀n m ht.Sem_HCL(DURING n m ht)  = during_sem n m(Sem_HCL_INST ht)) ∧
   (∀hi b.  Sem_HCL(WHEN hi b)      = when_sem(Sem_HCL hi)b) ∧
   (∀h1 h2. Sem_HCL(ANDc h1 h2)     = and_sem(Sem_HCL h1)(Sem_HCL h2)) ∧
   (∀h1 h2. Sem_HCL(THEN h1 h2)     = then_sem(Sem_HCL h1)(Sem_HCL h2) (End_HCL h1))
```

The semantic functions `Sem_HCL_INST` and `Sem_HCL` are defined in terms of semantic operators – one operator for each HCL constructor. For example, `is_sem` is defined as:

```
⊢def ∀i v. is_sem i value = (λstate. state i = value)
```

and `during_sem` is defined as:

```
⊢def ∀n m fn. during_sem n m fn = (λtj. ∀i. i < (m − n) ⇒ fn(tj(i + n)))
```

To illustrate the use of the above definitions, consider the HCL fragment:

```
    (‘in‘ is T) during (3,10)
```

Recall from Table 1 that a we model a trajectory as a function from natural numbers to states, and that a state is modeled as a function from node names to Boolean. Hence, using the above definitions we can prove that

```
⊢ Sem_HCL((‘in‘ is T) during (3,10))tj = (∀i. i < 7 ⇒ tj (i + 3) ‘in‘)
```

Intuitively, given a trajectory `tj`, i.e., an infinite sequence of circuit states, node `in` must be true in the circuit states $3, 4, \ldots, 9$ for this HCL formula to hold.

The definition of a formal semantics for HCL gives us a means of reasoning about HCL specifications at higher levels. We complement the above set of semantic functions with a corresponding set of compiler functions that can be used to automatically compile a HCL specification into a VOSS specification. After reading in a HCL specifications, these compiler functions can be applied to the parse tree representation of this HCL specification to generate a list of 5-tuples. The top-level compiler functions `Com_HCL_INST` and `Com_HCL` are defined as:

```
⊢def
   (         Com_HCL_INST UNC            = unc_com) ∧
   (∀s v.    Com_HCL_INST(IS s v)        = is_com s v) ∧
   (∀nl vl.  Com_HCL_INST(IS_VEC nl vl)  = is_vec_com nl vl)

⊢def
   (∀n ht.   Com_HCL(FOR n ht)          = for_com n(Com_HCL_INST ht)) ∧
   (∀n m ht.Com_HCL(DURING n m ht)      = during_com n m(Com_HCL_INST ht)) ∧
   (∀hi b.   Com_HCL(WHEN hi b)         = when_com(Com_HCL hi)b) ∧
   (∀h1 h2.  Com_HCL(ANDc h1 h2)        = and_com(Com_HCL h1)(Com_HCL h2)) ∧
   (∀h1 h2.  Com_HCL(THEN h1 h2)        = then_com(Com_HCL h1)(Com_HCL h2)(End_HCL h1))
```

where, for example, `unc_com` and `is_com` are defined as:

```
⊢def unc_com = (λb f d. [(F,‘‘,F,f,d)])
⊢def ∀n v. is_com n v = (λb f d. [(b,n,v,f,d)])
```

and `for_com` and `during_com` are defined as:

```
⊢def ∀d fn. for_com d fn = (λb f. fn b f(f + d))
⊢def ∀n m fn. during_com n m fn =
      (λb f. APPEND(for_com n unc_com b f)(fn b(f + n)(f + m)))
```

Note that the compiler function of an HCL program is a function of two arguments: the global domain constraint and the global start time. Normally these will be `T` and `0` respectively.

If we now return to our HCL fragment `(‘in‘ is T) during (3,10)` it is easy to prove that:

```
⊢ Com_HCL((‘in‘ is T) during (3,10))T 0 = [(F,‘‘,F,0,3); (T,‘in‘,T,3,10)]
```

which appears to be a reasonable translation.

It is now important to make sure that these compiler functions are correct. The problem of verifying our compiler function for compiling HCL into a list of 5-tuples is illustrated by the diagram in Fig. 2.

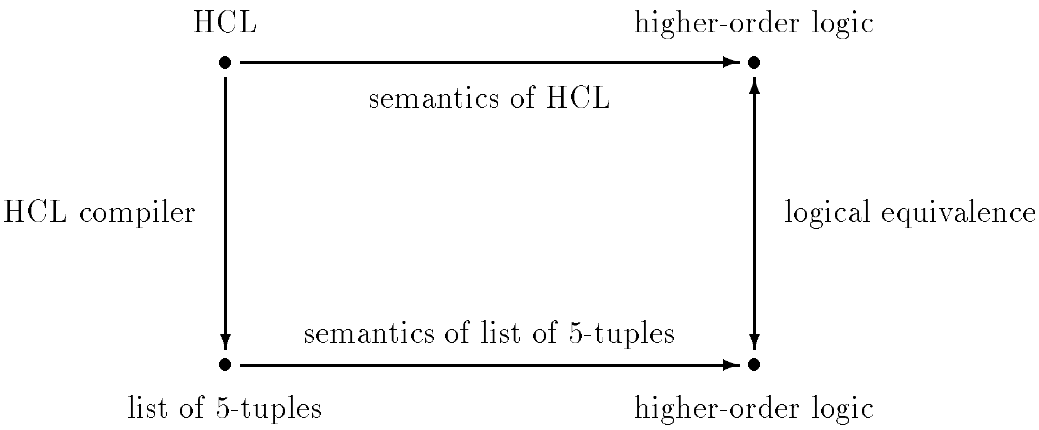

Figure 2: Verification of HCL compiler.

As suggested by the commutative diagram in Fig. 2, the compilation of any HCL program should result in a list of 5-tuples whose denotation (a term of higher-order logic) is logically equivalent to the denotation (also a term of higher-order logic) of the original HCL program. That is, we need to establish that

```
⊢ ∀hcl tj. Sem_Tuples (Com_HCL hcl T 0) tj = Sem_HCL hcl tj
```

where

```
⊢_def ∀l tj. Sem_Tuples l tj = ITLIST (λa b. a ∧ b) (MAP (Trajectory tj) l) T
```

We have in fact carried out such a compiler correctness proof for the HCL compiler.

Since the Voss system reads HOL definitions directly, the verified compiler functions are in fact the exact functions used to compile the HCL programs in the Voss system. Most examples of compiler verification only involve the verification of a compilation algorithm, but here we have verified the actual implementation of the compiler as a set of function definitions.

During the compiler proof we encountered a mistake in the definitions of the compiler functions. The oversight was related to the compilation of the `is_vec` construct. In the compiler functions we had simply assumed that the length of the node list was equal to the length of the value list, whereas the semantic functions make sure that this is indeed the case. A bug of this kind could quite conceivably cause the verification tool to generate a "false positive", i.e., declare a circuit to be correct despite that it contains an error. Thus we feel the time spent carrying out the compiler correctness result was well worth the effort. Also, the total time for a compiler proof for a language of this size is quite short (on the order of one person week). Finally, as for the bitvector arithmetic described earlier, the compiler proof is a task that is only done once.

6.3 Proof procedures

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 underlying results for these proof procedures are generalizations of 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. In this subsection we will highlight two of these general procedures.

The first procedure we will discuss deals with the generalization of timing specifications. As mentioned earlier, every finite-state machine specified by means of `new_fsm_specification` is suffix-closed — which is to say that every state is a possible initial state. As a consequence, it is easy to see that the set of trajectories for a given state machine is suffix closed, i.e., if $s = s_0, s_1, s_2, \ldots$ is a trajectory for some circuit, then any suffix of s is also a possible trajectory of the circuit. This observation allows us to prove the following result:

```
⊢ ∀P fsm. SuffixClosed fsm ⇒
       ((∀tj. fsm tj ⇒ (∀t. P(λn. tj(t + n)))) = (∀tj. fsm tj ⇒ P tj))
```

which essentially states that for any predicate over trajectories, `P`, if we need to establish that `P` holds for every start time `t`, then it is sufficient to prove that it holds for start time 0.

The above result allows us to develop a proof procedure, called `REMOVE_FORALL_TIME_TAC`, that can rewrite a goal of the form

```
∀v tj. fsm tj ⇒
       (∀t. Sem_HCL ((‘in‘ is v) during (t,t+5))tj ⇒
            Sem_HCL ((‘out‘ is ¬v) during (t+2,t+6))tj)
```

to a proof obligation like

```
∀v tj. fsm tj ⇒
       Sem_HCL ((‘in‘ is v) during (0,5))tj ⇒
       Sem_HCL ((‘out‘ is ¬v) during (2,6))tj
```

Our second example allows specifications to involve quantification over other domains besides Booleans. The Voss system can only quantify over Booleans. Hence, the highest level of specification that can be used in the Voss system must be stated in terms of bits and bitvectors. Unfortunately, we often would like to state our specifications in terms of quantifications over other domains than Booleans. For example, when dealing with circuits that are performing arithmetic operations, it is often more natural to state the correctness, and therefore the quantification, in terms of natural numbers. On the other hand, since Voss deals with an existing circuit of a specific size, we virtually always knows some bounds on the numbers we would like to quantify over. In other words, we would like to be able to quantify over some restricted, finite, domains other than Booleans.

The basic idea behind the RESTRICT_QUANT_TAC proof procedure is to replace quantification over a finite subset of the natural numbers with quantification over a list of Booleans used to represent the numbers. If "forall x::(f,t)" stands for "for all natural numbers x between f and t", then using the general theorem

```
⊢ ∀P f t.
      (forall x::(f,t). P x) =
      foralln (BITS_REQ(t − f)(t − f))
        (λbv. (num2bv(t − f)) bvgeq bv ⇒ P(bv2num(bv bvplus (num2bv f))))
```

where

```
⊢def ∀n P. foralln n P = foralln_aux n P[]

⊢def (∀P list. foralln_aux 0 P list = P list) ∧
     (∀n P list. foralln_aux(SUC n)P list = (∀a. foralln_aux n P(CONS a list)))
```

is a function that quantifies an expression over n Boolean variables and

```
⊢def (∀m. BITS_REQ 0 m = 0) ∧
     (∀n m. BITS_REQ(SUC n)m = ((m = 0) →  0 | SUC(BITS_REQ n(m DIV 2))))
```

denotes a function that computes the number of bits required to represent the larger number, the tactic RESTRICT_QUANT_TAC can be used to change a proof obligation like

```
forall i::(3,15). ∀tj. fsm tj ⇒ Sem_HCL(ant i)tj ⇒ Sem_HCL(cons i)tj
```

to the following proof obligation:

```
foralln
 (BITS_REQ(15 − 3)(15 − 3))
 (λbv.
   (num2bv(15 − 3)) bvgeq bv ⇒
   (∀tj.
     fsm tj ⇒
     Sem_HCL(ant(bv2num(bv bvplus (num2bv 3))))tj ⇒
     Sem_HCL(cons(bv2num(bv bvplus (num2bv 3))))tj))
```

If the antecedent and/or consequent now contains arithmetic functions using the argument i, we can then replace these arithmetic expressions with their corresponding bitvector version by applying the BV_ARITH_TAC discussed earlier. In the next section we will see an example of using these two proof procedures together.

A more interesting case is when we are not using these quantified values in arithmetic functions. Consider, for example a simple correctness proof of a 16 bit shift register. Intuitively, it is sufficient to check that the input is shifted in and that an arbitrary cell i is shifted to $i+1$ (assuming $i \leq 14$). Our high level specification might look like:

```
∀u v. forall i::(0,14).
   FSM
   shift_reg
   ( (cycles 2)
       andc ((‘In‘ is u) during (0,tau))
       andc (((EL i reg_nd_list) is v) during (0,10)),
     (((EL(i + 1)reg_nd_list) is v) during (tau,tau + 10))
       andc ((‘Q.0‘ is u) during (tau,tau + 10)))
```

Note that we are quantifying over `i`, but `i` is used to select a node in the node list. There are two alternatives to change this goal into something Voss can deal with. The easiest alternative is to simply let Voss explicitly perform the trajectory evaluation for $i = 0$, $i = 1$, $i = 2$, etc. up to $i = 14$. Unfortunately, if the circuit is complex, this can take a very long time. The alternative is to introduce a vector of Boolean variables to represent `i`. Call this bitvector `bi`. This in itself is, however, not sufficient since Voss is not capable of executing the HOL terms symbolically (it can only execute the circuit symbolically). So in our example, Voss could not compute `EL (bv2num bi) reg_nd_list`. However, we can perform the following algorithm: First compute for every instance of i that we want to quantify over the set of 5-tuples that would be used in the symbolic trajectory evaluation. Add the condition for each element in each set that corresponds to value j that `bi bvequal (num2bv j)`. Finally, take the union of all the sets of 5-tuples, run the symbolic trajectory evaluation using the complete sets, and finally quantify over the Boolean values in `bi`. In [2] this idea was introduced for a similar task and was there called symbolic indexing. In our version, we rely on the definition

```
⊢def (∀bv f.  symb_idx 0 bv f = (f 0) when (bv bvequal (num2bv 0))) ∧
     (∀n bv f.  symb_idx(SUC n)bv f =
            ((f(SUC n)) when (bv bvequal (num2bv(n + 1)))) andc (symb_idx n bv f))
```

and the fundamental theorem

```
⊢ ∀maxv m a c.
    (∀i. i ≤ maxv ⇒ FSM m(a i,c i)tl) =
    foralln
    (BITS_REQ maxv maxv)
    (λbi.
      (num2bv maxv) bvgeq bi ⇒
      FSM m(symb_idx maxv bi a,symb_idx maxv bi c)tl)
```

where `foralln` is defined as shown earlier and essentially quantifies a predicate over a list of Booleans. One important point to make is that this symbolic indexing theorem is very general. In particular, the dependency on i in the antecedent and consequent is left unspecified. In [2] symbolic indexing was used only for selecting a particular node in a fixed sized vector of nodes, which is a simple special case of the above theorem. Here, we have a very general result. Furthermore, since the result is provably correct (assuming we trust the semantic embedding of Voss in HOL of course), we can now use symbolic indexing (with confidence!) in many other instances. We believe that this type of general results that can be derived, given our semantic embedding of Voss in HOL, is an extremely important benefit from our approach.

An interesting note in this context is that there is now a new constraint imposed on how to write definitions in HOL if they are to be executed by Voss. Traditionally, definitions have been written as to simplify the later proofs, but now we must also be considering the execution speed of the functions. Consider, for example, the following two definitions of `POW2`

```
⊢def (∀m. POW2 0 = 1) ∧ (∀n. POW2 (SUC n) = 2 * (POW2 n))
⊢def (∀m. POW2 0 = 1) ∧ (∀n. POW2 (SUC n) = (POW2 n) + (POW2 n))
```

The two definitions are clearly logically equivalent. However, the first definition requires n function calls and n multiplications, whereas the second definition requires 2^n function calls, making it essentially unusable for anything but for very small n. We were "bitten" by this phenomena several times during the development of the symbolic indexing proof procedure and this is definitely an area that needs more work.

7 A Small Example

In this section we will illustrate the use of our two-level verification system on a small example. The main purpose of the example is to illustrate how the HOL-Voss system appears to the user. For a more thorough discussion how the HOL-Voss system can be used as a verification tool and for a number of larger examples, the reader is referred to [23].

Our example illustrates how a very high-level specification can be related to a very detailed model of a circuit. The circuit, shown in Fig. 3, is a simple Domino CMOS circuit (not necessarily a perfect example of Domino CMOS) with two 16 bits input words `a` and `b` and one output bit `out`. Intuitively, the circuit is supposed to compare the number presented on input `a` with the number presented on input `b` (both viewed as 16 bits unsigned binary numbers) and output 1 if and only if `a` $>$ `b` and `b` $\neq 0$, or, equivalently, 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, it is clear that the formal specification should be stated in terms of arithmetic relations and not in terms of bitvectors. However, at the same time, the circuit design uses quite complex electrical phenomena and critical timing and thus a fairly sophisticated switch-level and delay model is essential in order to explain the operation of the circuit.

With the above two requirements in mind, it becomes clear that no single verification tool is able to carry out the complete verification task in some reasonable amount of time. Consequently, it is an ideal illustration of the power of the HOL-Voss system. Here we will state the correctness result in terms of arithmetic relations and will quantify over a finite subset of the natural numbers. In particular, using the notation `forall` $x :: (a, b)$ to denote "for all numbers x greater than or equal to a and less than or equal to b", the high-level specification for the circuit is given in Fig. 4.

The complete proof script needed to prove this goal in the HOL-Voss system is shown in Fig. 5. The proof proceeds as follows. First we use the general theory for quantification over restricted domains as we described in Section 6.3 to replace the quantification over the natural numbers between 0 and 65353 by quantifications over two lists of Booleans. We then rewrite the proof obligations (the goal) by using the bitvector versions of the arithmetic relations. The third part of the proof deals with removing the "for all time" part of the specification and uses the proof procedure discussed in Section 6.3.

Since the Voss system uses Ordered Binary Decision Diagrams to represent the Boolean functions encountered during the verification, the efficiency and speed of the verification process depend highly on the ordering of the Boolean variables. Since it is inherently difficult to automatically derive a good variable ordering for Voss specifications (since they depend both on the circuit and the property we are trying to verify in a non-trivial way), we allow the user to give a variable order. It should be pointed out, however, that the ordering information can be incomplete and only will affect the efficiency of the verification task. In

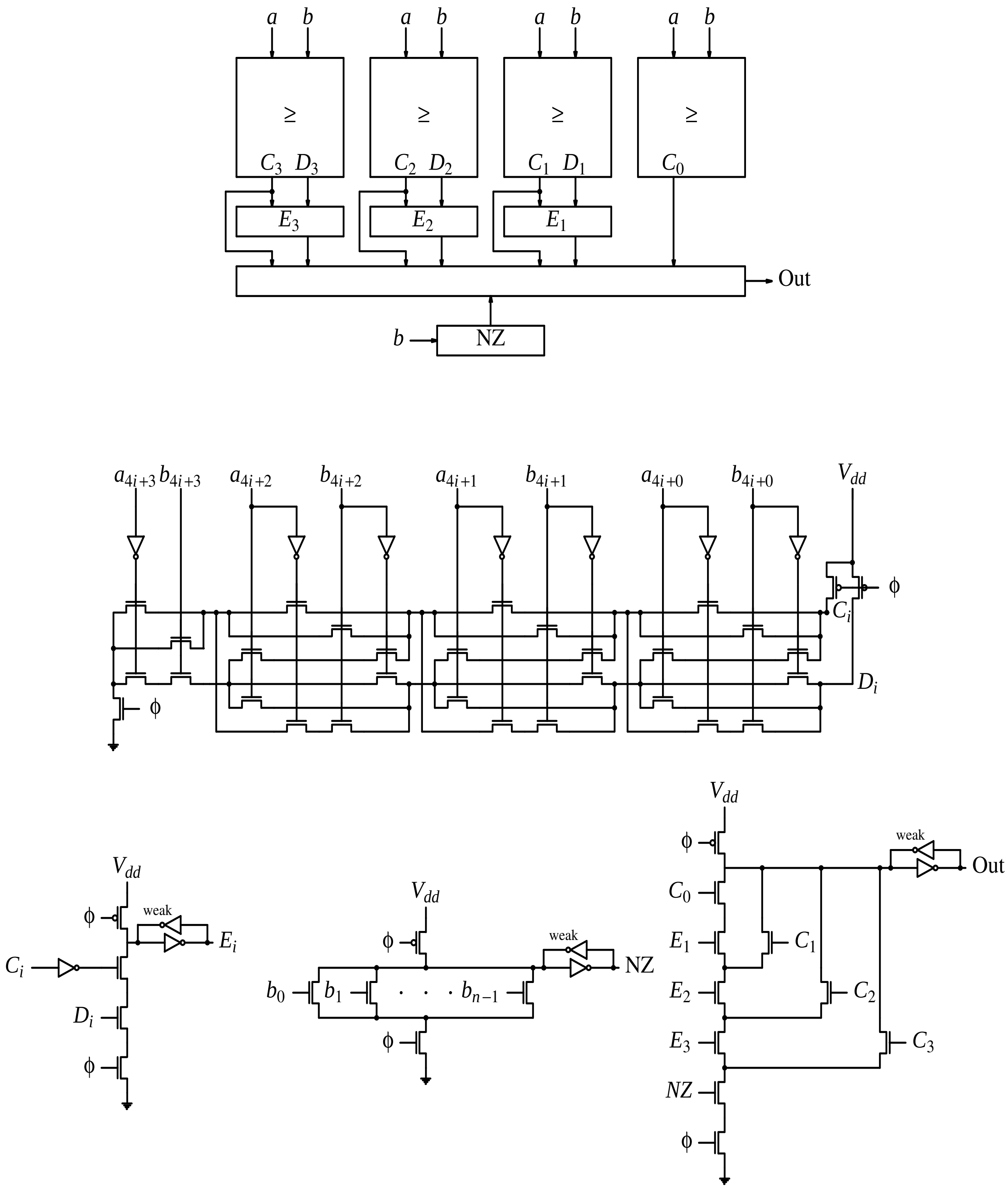

Figure 3: 16-bit circuit for computing $a > b > 0$.

```
loadt `../../hol_voss_init`;;

let AgrB16 = new_fsm_specification `AgrB16`;;

% Define shorthands for the node names %
let clock  = new_definition(`clock`, "clock = `phi1`");;
let output = new_definition(`output`, "output = `out`");;
let Na     = new_definition(`Na`, "Na = reverse (node_vec 16 `a`)");;
let Nb     = new_definition(`Nb`, "Nb = reverse (node_vec 16 `b`)");;

% The main verification goal %
g "forall a b::(0,65535). ∀(tj:trajectory). AgrB16 tj ⇒
   ∀(t:num).
     (Sem_HCL(
        ((clock is F) during (t,t+100)) andc
        ((clock is T) during (t+100,t+200)) andc
        ((Na is_vec (sized 16 (num2bv a))) during (t+80,t+200)) andc
        ((Nb is_vec (sized 16 (num2bv b))) during (t+80,t+200))) tj)
     ⇒
     (Sem_HCL ((output is ((a > b) ∧ (b > 0))) during (t+160,t+200)) tj)";;
```

Figure 4: High-level specification for AgrB circuit.

```
e RESTRICT_QUANT_TAC;;
e BV_ARITH_TAC;;
e REMOVE_FOR_ALL_TIME_TAC;;
e (ORDER_TAC "interleave (bv:bool list) (bv':bool list)");;
e (VOSS_TAC `AgrB16`);;
```

Figure 5: Proof script for AgrB circuit.

this case, a simple interleaving of the bitvector variables that correspond to the `a` and `b` variables is sufficient to achieve a reasonably short verification time.

Finally, the goal is sent to the Voss verification system by executing the `VOSS_TAC` tactic. In this case the verification is successful, and we obtain the final theorem:

```
⊢ forall a b::(0,65535).
    ∀tj.
     AgrB16 tj ⇒
     (∀t.
       Sem_HCL
       (((clock is F) during (t,t + 100)) andc
        (((clock is T) during (t + 100,t + 200)) andc
         (((Na is_vec (sized 16(num2bv a))) during (t + 80,t + 200)) andc
          ((Nb is_vec (sized 16(num2bv b))) during (t + 80,t + 200)))))
       tj ⇒
       Sem_HCL((output is (a > b ∧ b > 0)) during (t + 160,t + 200))tj)
```

Altogether, the complete verification takes about half a minute on a NeXT Station (25MHz 68040 processor). Most of this time is in fact spent loading the necessary libraries.

8 Conclusions and Future Work

Different methods of formal verification involve tradeoffs between automation, flexibility, expressibility, and accuracy. We conclude that a promising balance of these tradeoffs can be achieved by using theorem-proving at higher levels and symbolic trajectory evaluation at lower levels. Also, by integrating these two methods, we open up the possibility of verifying mixed software/hardware systems[3, 22].

We believe that this work represents one of the first successful attempts to develop a hybrid approach to formal hardware verification which is mathematically rigorous to ensure integrity, sufficiently general to be useful for a variety of design methodologies, and practical for achieving useful results that could not be easily obtained with existing verification tools. We are aware of previously published work done at IMEC in Belgium [17] 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-order logic. Also, the two-level verification tool described in [17] relies heavily on a specific design methodology in order to automate much of the proof obligations, whereas our goal is a general purpose verification system. Work at Edinburgh University by K. Goosens [16] also shares a common goal with our approach in the semantic embedding of a notion of verification based on symbolic simulation in a higher-order logic theorem prover. But unlike our approach where we have re-used an existing implementation of a symbolic simulation algorithm, Goosen has re-implemented a form of symbolic simulation as a proof rule in the Lambda theorem proving system.

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.

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. One possible avenue is the development of a richer specification language than HCL (in its present form) together with the definition of suitable semantic and compiler functions. We believe that some of our infrastructure which now exists in the form of proof procedures can be re-cast as features of a higher-level specification language leading to even greater automation of the theorem-proving process. As mentioned earlier, we also look forward to the possibility of other HOL-Voss users developing their own higher-level specification languages. It seems likely that development of higher-level specification languages for HOL-Voss can usefully build upon efforts by others to formalize conventional hardware description languages in higher-order logic [1, 4, 15, 30].

We are also considering the possibility of separating HOL-Voss into two tools. One of these tools would be a tool used exclusively by infrastructure builders with expertise in interactive theorem-proving. The main purpose of this expert tool would be to develop infrastructure—for example, the definition of a new bitvector arithmetic function or even the semantic embedding of a new higher-level specification language. Such infrastructure would be incorporated into a second standard "day-to-day tool" (i.e., Voss with extensive proof libraries) which would be used primarily for verifying circuits. Alternatively, we may elect to maintain HOL-Voss as a single tool but organize the functionality of this tool into a series of subsets. In the beginning a user would use only a very small subset of the functionality of HOL-Voss—perhaps nothing more than using HOL-Voss as a sophisticated simulator. As the user's expertise and confidence in the tool grows, the user can move on to increasingly large subsets with access to a greater range of verification techniques. This approach offers the advantage that very little re-training is needed when more sophisticated verification results are desired.

Acknowledgments

The authors are grateful to members of the ISD (Integrated Systems Design) research group at the University of British Columbia who have provided us with a stimulating environment for research in the area of formal methods and integrated systems. We are also grateful for the opportunity to discuss our methodology at the 1992 HOL User's Group meeting at IMEC in Leuven, Belgium where we received both encouragement and many helpful suggestions about both the presentation and substance of our work.

References

[1] C.M. Angelo, L. Claesen, and H. DeMan, "The Formal Semantics Definition of a Multi-Rate DSP Specification Language in HOL", *Proceedings of the HOL '92 Internation*

Workshop on Higher Order Logic Theorem Proving and its Applications, Elsevier, Amsterdam, expected 1993.

[2] Beatty, D.E., Bryant, R.E. and Seger, C-J., "Synchronous Circuit Verification—An Illustration", *Advanced Research in VLSI.* Proceedings of the Sixth MIT Conference, ed. William Dally, pp.98-112, MIT Press, Cambridge MA, 1990.

[3] W. Bevier, W. Hunt, J Moore, and W. Young, "An Approach to Systems Verification", *Journal of Automated Reasoning*, Vol. 5, No. 4, November 1989.

[4] R. Boulton, M. Gordon, J. Herbert and J. Van Tassel, "The HOL Verification of ELLA Designs", in: P. Subrahmanyam, ed., *Proceedings of a Workshop on Formal Methods in VLSI Design*, 9–11 January 1991, Miami, Florida.

[5] R. S. Boyer and J.S. Moore, *A Computational Logic Handbook*, Academic Press, 1988.

[6] 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.

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

[8] 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.

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

[10] R.E. Bryant, and C-J. Seger, "Formal Verification of Digital Circuits Using Symbolic Ternary System Models", *DIMAC Workshop on Computer-Aided Verification*, Rutgers, New Jersey, June 18-20, 1990 (to appear in Springer Verlag's Lecture Notes in Computer Science).

[11] Albert John Camilleri, "Mechanizing CSP Trace Theory in Higher Order Logic", *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 9, September 1990, pp. 993-1104.

[12] W.C. Carter, W.H. Joyner, Jr., and D. Brand, "Symbolic Simulation for Correct Machine Design", *16th ACM/IEEE Design Automation Conference*, 1979, pp. 280–286.

[13] Avra Cohn, "The Notion of Proof in Hardware Verification", *Journal of Automated Reasoning*, Vol. 5, May 1989, pp. 127-139.

[14] Paul Curzon, "Deriving Correctness Properties of Compiled Code", *Proceedings of the HOL '92 Internation Workshop on Higher Order Logic Theorem Proving and its Applications*, Elsevier, Amsterdam, expected 1993.

[15] K.G.W. Goosens, "Embedding a CHDDL in a Proof System", Technical Report No. ECS-LFCS-91-155, Department of Computer Science, University of Edinburgh, May 1991.

[16] K.G.W. Goosens, "Operational Semantics Based Formal Symbolic Simulation", *Proceedings of the HOL '92 Internation Workshop on Higher Order Logic Theorem Proving and its Applications*, Elsevier, Amsterdam, expected 1993.

[17] Mark Genoe, Luc Claesen, Eric Verlind, Frank Proesmans, and Huge 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.

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

[19] M.J.C. Gordon and T.F. Melham (eds.), "Introduction to HOL", Cambridge University Press, expected 1993.

[20] Roger W. S. Hale, "Reasoning about Software", in *HOL 91*, Proc. of the 1991 Int. Workshop on the HOL Theorem Proving System and its Application, Eds. M Archer, J. Joyce, K. Levitt, and P. Windley, IEEE Computer Society Press, 1991, pp. 52–58.

[21] Jeffrey J. Joyce, "A Verified Compiler for a Verified Microprocessor", Report No. 167, Computer Laboratory, Cambridge University, March 1989.

[22] Jeffrey J. Joyce, "Totally Verified Systems: Linking Verified Software to Verified Hardware", in: *Specification, Verification and Synthesis: Mathematical Aspects*, Proceedings of a Workshop, 5-7 July 1989, M. Leeser and G. Brown, eds., Ithaca, N.Y., Springer-Verlag, 1989.

[23] J. J. Joyce and C-J. Seger, "Using the HOL-Voss System for Hardware Verification", in preparation.

[24] E. Mendelson, *Introduction to Mathematical Logic*, D. Van Nostrand Company, Inc., Princeton, N.J., 1964.

[25] S. Mirapuri, M. Woodacre, and N. Vasseghi, "The Mips R4000 Processor", *IEEE Micro*, April 1992, pp. 10-22.

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

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

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

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

[30] John P. Van Tassel, "A Formalisation of the VHDL Simulation Cycle", *Proceedings of the HOL '92 Internation Workshop on Higher Order Logic Theorem Proving and its Applications*, Elsevier, Amsterdam, expected 1993.