A Two-Level Formal Verification Methodology using HOL and COSMOS

by

Carl-Johan Seger
and
Jeffrey J. Joyce

Technical Report 91-10
June, 1991

Department of Computer Science
University of British Columbia
Vancouver, B.C.
CANADA V6T 1Z2

A Two-Level Formal Verification Methodology using HOL and COSMOS.*

Carl-Johan Seger and Jeffrey J. Joyce
Department of Computer Science
University of British Columbia
Vancouver, B.C., CANADA V6T 1W5

e-mail: seger@cs.ubc.ca
joyce@cs.ubc.ca

June 18, 1991

Abstract

Theorem-proving and symbolic simulation 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—and its most original contribution—is the suggestion that symbolic simulation and theorem-proving can be combined in a complementary manner. We also outline our plans for the development of a mathematical interface between the two approaches—in particular, a semantic link between the formulation of higher-order logic used in the Cambridge HOL system and the specification language used in the COSMOS system. We believe that this combination offers great potential as a practical formal verification methodology which combines the ability to accurately model circuit level behavior with the ability to reason about digital hardware at higher levels of abstraction.

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. Despite major simulation efforts, serious design errors often remain undetected. Consequently, there has been a growing interest in using formal methods to verify the correctness of designs. There are several approaches to formal hardware verification: theorem-proving, state machine analysis, and symbolic simulation to mention a few. 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.

*This research was supported by operating grants from the Natural Sciences and Engineering Research Council of Canada.

Most research on formal verification has relied on the use of computer-assisted theorem provers [3, 8, 9, 12, 16] to establish equivalence between different circuit representations. Here the circuit is described hierarchically, where a component is defined at one level in the hierarchy as an interconnection of components defined at lower levels. The system specification consists of behavioral descriptions of the components at all levels in the hierarchy. Verification involves proving that each component fulfills its part of the specification, assuming that its constituent components fulfill their specifications. This proof is mostly carried out interactively using some kind of computer-assisted theorem prover, like the Boyer-Moore Theorem Prover [3] or the Cambridge HOL System [16].

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. 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.

A first-generation symbolic simulator resembles a traditional logic simulator [5]. The simulator computes how a circuit would behave in response to a sequence of input patterns. These patterns, however, can contain *Boolean variables* in addition to the constants 0 and 1. Consequently, the results of the simulation are 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.

A verifier based on symbolic simulation applies logic simulation to compute the circuit's response to a series of stimuli chosen to detect all possible design errors. When a circuit has been "verified" by simulation, this means that any further simulation would not uncover any errors. Since a symbolic simulator is based on a traditional logic simulator, it is 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 will link the physical layout with some higher level of specification.

Recently, Bryant and Seger [6] developed a second-generation symbolic simulator. Here the simulator establishes the validity of formulas expressed in a very limited, but precisely defined, temporal logic. By limiting the complexity of the logic, great efficiency is obtained. Furthermore, the verification process is highly automated. Unfortunately, the automation obtained by the symbolic simulator 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. Secondly, the semantic gap between the intuitive, informal, specification the designer has in mind and the specification used in the symbolic simulator is often quite large.

When tabulating the strengths and weaknesses of theorem-proving and symbolic simulation used for formal hardware verification, it is striking to see how well the two approaches complement each other. Thus, it is very appealing to attempt to integrate them into a two-level combined approach to formal hardware verification. However, in order to achieve this integration, two problems need to be resolved: 1) a mathematically precise interface must be developed so that the rigor of the formal proof is not jeopardize, and 2) a practical interface between the two processes must to be developed. In this paper we focus on the first issue and only briefly mention ongoing work towards solving the

second problem.

We believe that the combination of general-purpose theorem-proving and symbolic simulation offers great potential as a practical formal verification methodology: it combines the ability to reason about detailed circuit level behaviour accurately and efficiently with the ability to reason about digital hardware at higher levels of abstraction. To the best of our knowledge, this is the first instance of when theorem-proving and symbolic simulation have been used in a combined approach to formal hardware verification.

2 A Two-Level Approach

Symbolic simulation, as achieved by the COSMOS simulator, can be viewed as a highly specialized form of theorem-proving. COSMOS checks the validity of assertions in a specification language (which we call *CL*) with respect to a model structure Ψ. This model structure is a set of infinite state sequences determined by an extracted circuit netlist $\mathcal{C}$ and a built-in switch-level and delay model of circuit behaviour. When viewed as a theorem-proving system, the COSMOS system can be used to prove theorems of the form, $\Psi \models \mathtt{f}$, where $\mathtt{f}$ is a formula in *CL* .

A rigorous link with general-purpose theorem-proving, in particular, the Cambridge HOL system, is achieved by semantically embedding the specification language *CL* in higher-order logic. The semantic embedding of *CL* in higher-order logic allows *CL* specifications to be expanded into a term of higher-order logic and used to derive higher-level correctness results. That is, the HOL system can be used to prove theorems of the form, $\vdash (\Psi \models \mathtt{f}) \Longrightarrow \mathtt{t}$, where $\mathtt{f}$ is a formula in *CL* (embedded in higher-order logic) and $\mathtt{t}$ is a term in higher-order logic.

Thus, the two proof results, $\Psi \models \mathtt{f}$ and $\vdash (\Psi \models \mathtt{f}) \Longrightarrow \mathtt{t}$ constitute a statement of correctness in our combined approach. They are obtained by symbolic simulation and general-purpose theorem-proving respectively.

In Section 3, we elaborate on how symbolic simulation can be viewed as a highly specialized form of theorem-proving. Section 4 explains how *CL* can be embedded in higher-order logic to establish a rigorous link between the two approaches. Finally, in Section 5, we describe how CL is used to bootstrap a higher level specification language, called HCL.

3 Symbolic Simulation Viewed as Theorem-Proving

The main thesis of this section is that the verification system described in [6], based on the COSMOS symbolic simulator, can be viewed as a proof system. In fact, the system can be viewed as proving that the behaviors derived from an extracted netlist imply certain properties described in a formal specification language. We use the simple example of an inverter, Fig. 1(a), to provide the reader with an informal account of our approach. If we assume a binary circuit model and a unit delay simulator, the inverter circuit is accurately described by the state machine shown in Fig. 1(b). The states of the machine are labelled with the current values of the two nodes in the circuit, and a transition in the state machine corresponds to a basic unit of time.

The state machine in Fig. 1(b) implies certain properties. For example, it is easy to see that we can conclude from the state machine that the value on the output is always the complement of the value that was present on the input one time unit ago. Informally, this could be written as:

for every state sequence $\quad [(\mathtt{in} = a) \Longrightarrow \mathbf{X}(\mathtt{out} = \overline{a})]$

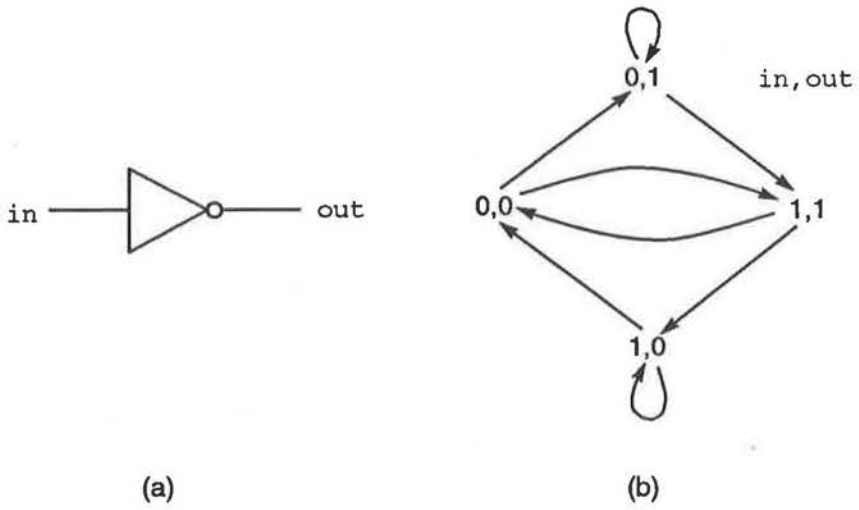

Figure 1: Inverter and corresponding state machine.

where the **X** is a "next time" operator and $\overline{a}$ denotes the Boolean complement of a. The main result of [6] is that the COSMOS symbolic simulator can be used to prove this kind of statement. In other words, the COSMOS system can be used to prove that the behaviors derived from an extracted netlist using a sophisticated switch-level and timing model implies certain formulas described in a logic with precisely defined semantics.

3.1 Circuit State Machine and Trajectories

The circuit model used in [6] is a ternary model, i.e., nodes in the circuit can take on the values 0, 1, and X. The *circuit state machine* corresponding to some circuit C is a non-deterministic finite state machine $\mathcal{M} = (S, \Delta, \Theta)$, where S is a finite set of states, Δ, the *transition relation*, is a relation on S, and Θ is a function $\Theta: S \rightarrow \{0, 1, X\}^n$ relating every state in S to an assignment of 0s, 1s, and Xs to the nodes of the circuit. Intuitively, if the circuit currently is in the state s^i and $(s^i, s^{i+1}) \in \Delta$, then the circuit can be in the state s^{i+1} one basic time unit later.

The circuit state machine is determined by three factors: 1) the extracted netlist, 2) the switch-level model, and 3) the delay model. Ideally, the netlist is obtained by netlist extraction from a mask-level representation of the circuit. The switch-level model, in our COSMOS based approach, is the MOSSIM II [4] switch-level model. COSMOS provides a variety of delay models including unit-delay, nominal-delay, bounded-delay, and arbitrary-delay models.

Given a circuit state machine, a *state trajectory*, χ, is an infinite sequence of states $s^1, s^2, \ldots$, such that $s^i \in S$ and $(s^i, s^{i+1}) \in \Delta$ for $i \geq 1$. The *circuit trajectory*, $\psi(\chi)$, corresponding to a state trajectory χ is an infinite sequence of ternary state vectors $a^1, a^2, \ldots$, such that $a^i \in \{0, 1, X\}^n$ and $a^i = \Theta(s^i)$ for $i \geq 1$. Informally, a circuit trajectory can be viewed as an infinite sequence of "snapshots" of the operating circuit taken every unit of time. Finally, let Ψ denote the set of all possible circuit trajectories for a given circuit. Intuitively, Ψ can be viewed as the set of all possible circuit behaviors according to the switch-level and delay model used.

3.2 Logic *CL*

The logic *CL* is defined in terms of another logic called *CL′* . We begin by describing *CL′* and then consider *CL*.

The logic *CL′* is defined over a set of nodes, $\mathcal{N} = \{n_1, \ldots, n_n\}$, and over a set of symbolic Boolean variables, $\mathcal{V}$. The formulas consists of constants (TRUE), atomic propositions ($n_i = 1$ and $n_i = 0$), conjunction ($\mathtt{f}_1 \wedge \mathtt{f}_2$), case restriction ($e \rightarrow \mathtt{f}$), and next time operations ($\mathbf{X}\mathtt{f}$). In case restriction, ($e \rightarrow \mathtt{f}$), e is a Boolean expression over $\mathcal{V}$ and $\mathtt{f}$ is a *CL′* formula. The basic idea is to use a Boolean function to limit the cases for which the *CL′* formula $\mathtt{f}$ is of interest. For more details, see [6].

Let $\mathcal{V}$ be a set of symbolic Boolean variables. An *interpretation*, ϕ, is a function $\phi: \mathcal{V} \rightarrow \mathcal{B}$ assigning a binary value to each symbolic Boolean variable. Let Φ be the set of all possible interpretations, i.e., $\Phi = \{\phi: \mathcal{V} \rightarrow \mathcal{B}\}$.

The truth semantics of a *CL′* formula $\mathtt{f}$ is defined relative to an interpretation $\phi \in \Phi$ and a circuit trajectory $\psi = a^1, a^2, a^3, \ldots \in \Psi$. For a precise definition of the truth semantics, see [6]. Informally, the *CL′* formula TRUE holds for every ϕ and ψ. The formula $n_i = 0$ ($n_i = 1$) holds if and only if $a_i^1 = 0$ ($a_i^1 = 1$). The conjunction of two *CL′* formulas holds if and only if both formulas hold. The *CL′* formula $e \rightarrow \mathtt{f}$ holds if either the Boolean formula denoted by the Boolean expression e evaluates to 0 for interpretation ϕ, or if the *CL′* formula $\mathtt{f}$ holds. Finally, $\mathbf{X}\mathtt{f}$ holds for ϕ and circuit trajectory $a^1, a^2, a^3 \ldots \in \Psi$ if and only if $\mathtt{f}$ holds for ϕ and circuit trajectory $a^2, a^3 \ldots$.

The verification methodology used by the COSMOS system entails proving *assertions* about the model structure. These assertions, written in the *core logic CL* , are of the form

$$\mathtt{A} \implies \mathtt{C},$$

where the *antecedent* $\mathtt{A}$ and the *consequent* $\mathtt{C}$ are *CL′* formulas over $\mathcal{N}$ and $\mathcal{V}$. This assertion is true, written $\Psi \models (\mathtt{A} \implies \mathtt{C})$, if and only if for every interpretation, i.e., every assignment of 0s and 1s to the symbolic Boolean variables, and for every possible circuit trajectory, the *CL′* formula $\mathtt{C}$ holds or the *CL′* formula $\mathtt{A}$ does not hold.

3.3 A Decision Algorithm

A decision algorithm based on ternary symbolic simulation was given in [6] for determining the validity of formulae in *CL* . That is, the algorithm determines whether or not for every interpretation every circuit trajectory satisfying the antecedent A must also satisfy the consequent C. It does this by generating a symbolic simulation sequence corresponding to the antecedent, and testing whether the resulting symbolic state sequence satisfies the consequent. For details, see [6].

4 Semantically Embedding *CL* in Higher-Order Logic

As argued in [14], a major advantage of higher-order logic as a formalism for verifying hardware is the ability to semantically embed more specialized formalisms into this logic. This often results in more concise specifications and easier proofs. Furthermore, as this paper demonstrates, the ability to semantically embed another formalism in higher-order logic, in particular *CL* , provides a means of establishing a rigorous link between general-purpose theorem-provers, such as HOL , and other verification tools, such as COSMOS .

A variety of formalisms have been semantically embedded in the HOL logic. This includes programming logics [10], temporal logics [11], process algebras [7] and subsets of conventional hardware description languages [2].

Although full details including machine-readable syntax are beyond the scope of this paper, a sketch of how *CL* can be embedded in higher-order logic is given below. As explained earlier, the truth semantics of a *CL'* formulae is relative to an interpretation ϕ and a circuit trajectory ψ. Thus, operators of *CL'* are defined as functions of ϕ and ψ. An interpretation ϕ is represented as a function that maps Boolean expressions (of *CL'*) to Boolean values. Circuit trajectories are also represented by functions—in this case, functions that map position in a sequence to state vectors.

$\vdash_{def}$ TRUE = $\lambda\ \phi\ \psi$. T

$\vdash_{def}$ f1 $\wedge$ f2 = $\lambda\ \phi\ \psi$. (f1 $\phi\ \psi$) $\wedge$ (f2 $\phi\ \psi$)

$\vdash_{def}$ e $\rightarrow$ f = $\lambda\ \phi\ \psi$. (ϕ(e)) $\Longrightarrow$ (f $\phi\ \psi$)

$\vdash_{def}$ **X**f = $\lambda\ \phi\ \psi$. f ϕ (tail(ψ))

We also need to embed the *CL* operator $\Longrightarrow$ in higher-order logic and provide a notion of validity for *CL* formulae. Suitable definitions are characterized by the following theorem.

$\vdash_{thm}$ $\Psi \models$ A $\Longrightarrow$ B = $\forall\phi \in \Phi, \forall\psi \in \Psi$. (A $\phi\ \psi$) $\Longrightarrow$ (C $\phi\ \psi$)

To avoid a proliferation of symbols in this informal account, we have used the same symbol for conjunction in both *CL'* and higher-order logic, namely, $\wedge$. Likewise, the symbol $\Longrightarrow$ is used in both *CL* and higher-order logic. However, these are truly different operators (with a different logical status). The actual definitions entered into the HOL system introduce new symbols, i.e., other than the built-in HOL constants $\wedge$ and $\Longrightarrow$.

5 A Higher-Level Intermediate Language

Earlier, in Section 2, we described how *CL* can be used as an intermediate specification language which provides the basis for a rigorous link between COSMOS and HOL . In fact, this is a simplified (but not inaccurate) view of our approach. It turns out that *CL* is too low level to serve as an intermediate specification language between COSMOS and HOL . We actually use a higher level specification language called *HCL* .

HCL is a more concise language for writing specifications of circuit behaviour. For instance, certain functions performed by arithmetic hardware would be very tedious to write directly in *CL*—but they can be expressed very succinctly in *HCL* using recursion.

The COSMOS system compiles *HCL* specifications into *CL* specifications. In the simplified view of our approach presented earlier in Section 2, we described how a rigorous link between HOL and COSMOS could be achieved by semantically embedding *CL* in higher-order logic. But the use of *HCL* , rather than *CL* , as an intermediate specification language results in a weak link (i.e., not rigorous) between HOL and COSMOS in our current implementation. We intend to make this link rigorous in the near future by:

1. Defining the formal syntax of *HCL* .

2. Semantically embedding *HCL* in higher-order logic.

3. Formally specifying an algorithm for compiling *HCL* into *CL* .

4. Formally verifying that this compiling algorithm is correct.

The problem of verifying of an algorithm for compiling *HCL* into *CL* is illustrated by the diagram in Fig. 2.

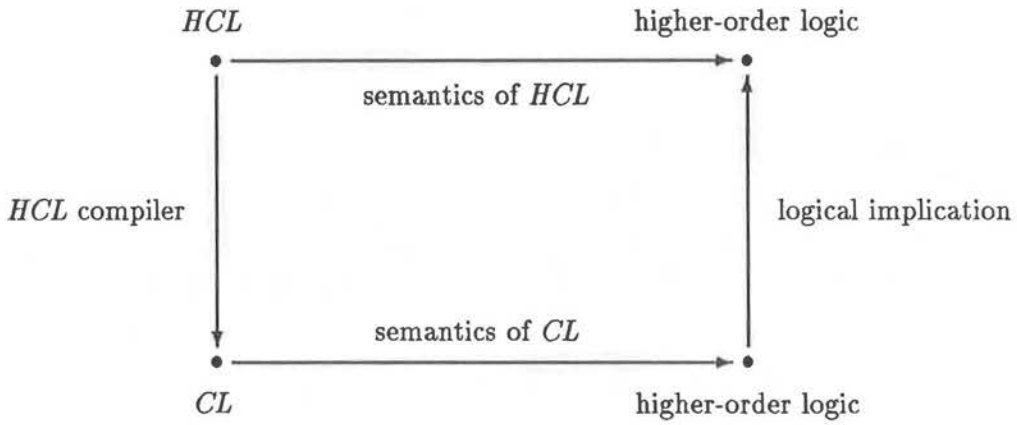

Figure 2: Verification of *HCL* compiler.

As suggested by the commutative diagram in Fig. 2, the compilation of any *HCL* specification should result in a *CL* specification whose denotation (a term of higher-order logic) logically implies the denotation (also a term of higher-order logic) of the original *HCL* specification. That is,

$$\forall \mathtt{f} \in \mathit{HCL}\ \ \mathtt{SemCL}\ (\mathtt{CompHCL}\ (\mathtt{f}\ )) \Longrightarrow \mathtt{SemHCL}\ (\mathtt{f}\ )$$

where SemCL and SemHCL are semantics functions for *CL* and *HCL* respectively and CompHCL is compiling function for *HCL* , i.e., a specification of the compiling algorithm. Formal proof of the above theorem is well within the scope of the HOL system; for an example of compiler verification using the HOL system, see [13, 15].

Verifying an algorithm for compiling *HCL* into *CL* would establish a rigorous link between HOL and COSMOS — to the extent that we trust that the compiling algorithm can be correctly implemented (as part of the COSMOS system). The use of *HCL* , instead of just *CL* , as the intermediate specification language, means that a statement of correctness in our approach consists of two results, $\Psi \models \mathtt{f}$ and $\vdash (\Psi \models \mathtt{f}\ ) \Longrightarrow \mathtt{t}$ where f is formula of *HCL* (rather than a formula of *CL*) and t is a term of higher-order logic. As before, these results would be obtained by symbolic simulation and general-purpose theorem-proving respectively.

6 An Example

To illustrate our two-level approach to formal hardware verification, consider the circuit shown in Fig. 3. This is a 16-bit instance of a (pseudo) domino-logic design for a circuit that tests whether:

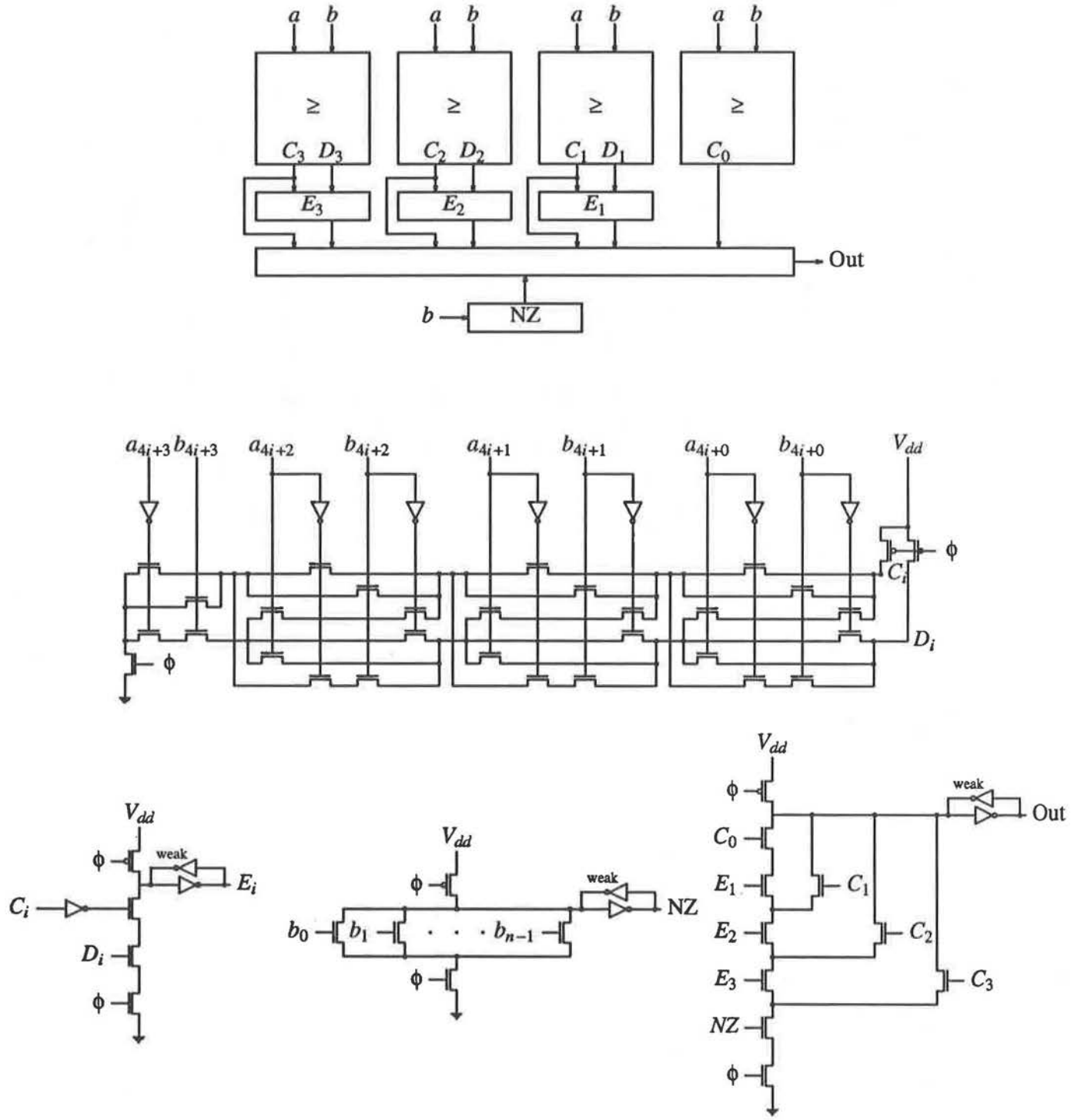

Figure 3: 16-bit circuit for computing $A > B > 0$.

1) input A is greater than input B and, 2) input B is greater than zero, when these inputs are interpreted as the unsigned binary representation of two numbers. The goal of formal verification is to relate a top-level specification of this circuit's intended function to a bottom-level specification of its implementation (based on an underlying model of hardware). The top-level specification should be sufficiently abstract to minimize the semantic gap between it and the informal, intuitive, specification of the circuit that resides in the mind of the designer. On the other hand, the bottom-level specification should be an accurate model of the circuit. This includes not only an accurate electrical model but also temporal properties of the circuit.

In the mind of the human specifier, the intended function of the circuit shown in Fig. 3 is intuitively understood in terms of an arithmetic relation, i.e., "the output should be 1 iff A is greater than B and B is greater than 0". To minimize the semantic gap, the top-level formal specification should also be stated in terms of an arithmetic relation. At the bottom-level of specification, the actual operation of the circuit shown in Fig. 3 cannot be accurately described by a simple model of circuit behaviour. A number of detailed features such as clocking, charge storage, charge sharing, and sized transistors, need to be included in an accurate model of this circuit. Hence, the verification problem, in this particular case, is to relate a top-level specification expressed in terms of an arithmetic relation to a bottom-level specification based on a detailed model of switch level circuit behaviour.

Neither symbolic simulation or theorem-proving is able to satisfactorily deal with this verification problem. Symbolic simulation would clearly be unable to support a top-level specification stated in terms of arithmetic relations. Theorem-proving is generally inappropriate for reasoning about detailed circuit behaviour. Below, we will outline how this proof could be carried out using our combined verification approach.

A behavioural specification expressed in *HCL* is given below for the circuit in Fig. 3. These definitions are written in a syntax based on the C programming language. This is because *HCL* is currently just an extension of C—but this will soon be replaced by a rigorous definition of the *HCL* language when *HCL* is semantically embedded in higher-order logic.

```
Bool greater (int n, Bool *a_vec, Bool *b_vec)
{
    if( n == 0 ) return( Zero() ); else
         return( Or( And(a_vec [n-1], Not(b_vec [n-1])),
                     And(Equal (a_vec [n-1], b_vec [n-1])),
                         greater(n-1,a_vec, b_vec)) );
}

Bool notzero (int n, Bool *x_vec)
{
    if( n == 0 ) return( Zero() ); else
       return( Or(x_vec [n-1], notzero (n-1, x)) );
}

Bool CMP_BitLevel (int n, Bool *a_vec, Bool *b_vec)
{
    return( And(greater(n, a_vec, b_vec), notzero(n, b_vec)) );
}


HCL Timing (int n, Bool *a_vec, Bool *b_vec, Bool out_val)
{
     return( Imply( Conj( During(0, 100, Is(phi, Zero())),
                          During(100, 200, Is(phi, One())),
                          During(95, 200, Is(a, a_vec)),
                          During(95, 200, Is(b, b_vec))),
                    During(180, 200, Is(out, out_val))));
}
```

The above definition of `CMP_BitLevel` specifies the bit level "compare operation" implemented by the circuit in Fig. 3. The definition of this operation is parameterized by the size of the circuit—in the case of the 16-bit version of this circuit shown in Fig. 3, `n` would be assigned the value 16.

`Timing` describes the timing conditions under which we wish to verify that the circuit in Fig. 3. To paraphrase this definition: on the assumption that,

- the clock signal `phi` is `Zero ()` (i.e., low) for 100 time units and then is `One ()` (i.e., high) for another 100 time units,
- the vectors of circuit nodes `a` and `b` are assigned the vectors of symbolic Boolean variables `a_vec` and `b_vec` at time 95 and held stable until time 200,

then the circuit node denoted by `out` must be equal to the value `out_val` from at least time 180 until time 200.

The functional specification expressed by `CMP_BitLevel` and the timing conditions expressed by `Timing` are combined in the top level *HCL* specification shown below.

```
Timing (n, a_vec, b_vec, CMP_BitLevel (n, a_vec, b_vec))
```

The COSMOS system is able to derive the following theorem which states that the above *HCL* specification (with `n` instantiated as `16`) is a logical consequence of the finite state machine derived from the extracted netlist of the circuit shown in Fig. 3.

```
Ψ ⊨ Timing (16, a_vec, b_vec, CMP_BitLevel (16, a_vec, b_vec))
```

We now wish to derive a more abstract correctness result which expresses correctness at the arithmetic level. Currently, we must hand-translate *HCL* specifications into higher-order logic. Eventually, when *HCL* is semantically embedded in higher-order logic, this translation will be a series of expansion steps governed by the inference rules of higher-order logic. This series of expansion steps will be mechanically checked (and largely automated) by the HOL system.

To formally establish a relationship between a bit level correctness result and a higher level correctness result expressed in terms of natural number arithmetic, we need to formally define a relationship between bit vectors and natural numbers. This is expressed by the definition of BitsToNum which is a data abstraction function that maps bit vectors to natural numbers.

```
⊢def BitsToNum (n,x_vec) =
       if (n = 0) then 0 else
         (((x_vec [n-1]) => (2^(n-1)) | 0) + BitsToNum (n-1,x_vec)))
```

We also define the function CMP_NumLevel which is an arithmetic level specification of the function performed by the circuit in Fig. 3.

```
⊢def CMP_NumLevel (a_num,b_num) =
       if (b_num > 0) then (a_num > b_num) else false
```

The HOL system can be used to prove:

```
⊢thm ∀n a_vec b_vec.
        CMP_BitLevel(n,a_vec,b_vec) ≡
        CMP_NumLevel(BitsToNum (n,a_vec),BitsToNum (n,b_vec))
```

Having established this equivalence, we can then derive a generalized correctness result which relates the bit level specification of the compare circuit to an arithmetic level specification for any value of n:

```
⊢thm ∀n .
       Ψ ⊨ Timing (n, a_vec, b_vec, CMP_BitLevel (n,a_vec,b_vec))))
       ⟹
       Ψ ⊨ Timing (n, a_vec, b_vec,
              CMP_NumLevel (BitsToNum (n,a_vec),BitsToNum (n,b_vec)))
```

Finally, we instantiate this generalized result for n = 16 to obtain,

```
⊢thm Ψ ⊨ Timing (16, a_vec, b_vec, CMP_BitLevel (16,a_vec,b_vec))))
     ⟹
     Ψ ⊨ Timing (16, a_vec, b_vec,
                 CMP_NumLevel (BitsToNum (16,a_vec),BitsToNum (16,b_vec)))
```

which, together with the symbolic simulation result,

$\Psi \models$ `Timing (16, a_vec, b_vec, CMP_BitLevel (16, a_vec, b_vec))`

constitutes a statement of correctness for the circuit in Fig. 3.

7 Conclusions

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 simulation at lower levels. By embedding the "high-level" specification logic used by COSMOS into HOL , we are able to efficiently verify systems from a very detailed electrical and timing domain up to a very abstract behavioral domain.

We think that this two-level approach will be particularly useful in the case of circuits where there is tight coupling between functional and temporal properties of the circuit level and high level abstractions, e.g., when a gate level or RTL abstraction is not available as an intermediate level. This is especially true in the case of high performance designs. Also, by integrating these two methods, we open up the possibility of verifying mixed software/hardware systems[1, 15].

We are currently in the process of formalizing the HCL logic and implementing an HCL compiler in the COSMOS system. This involves not only modifying the existing, informal, HCL compiler in COSMOS , but also to define the precise semantics of the language and proving the correctness of the compilation method.

References

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

[2] 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.

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

[4] 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.

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

[6] 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).

[7] 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.

[8] Paolo Camurati and Paolo Prinetto, "Formal Verification of Hardware Correctness", *IEEE Computer*, Vol. 21, No. 7, July 1988, pp. 8-19.

[9] M. J. C. Gordon, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware", in: G. Milne and P. Subrahmanyam, eds., *Formal Aspects of VLSI Design*, Proceedings of the 1985 Edinburgh Conference on VLSI, North-Holland, 1986, pp. 153-177.

[10] Michael J. C. Gordon, "Mechanizing Programming Logics in Higher Order Logic", in: G. Birtwistle and P. Subrahmanyam, eds., *Current Trends in Hardware Verification and Automated Theorem Proving*, Springer-Verlag, 1989, pp. 387-439. Also Report No. 145, Computer Laboratory, Cambridge University, September 1988.

[11] Roger W. S. Hale, *Programming in Temporal Logic*, Ph.D. Thesis, Report No. 173, Computer Laboratory, Cambridge University, July 1989.

[12] Warren A. Hunt, *FM8501, A Verified Microprocessor*, Ph.D. Thesis, Report No. 47, Institute for Computing Science, University of Texas, Austin, December 1985.

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

[14] Jeffrey J. Joyce, "More Reasons Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware", in: P. Subrahmanyam, ed., *Proceedings of a Workshop on Formal Methods in VLSI Design*, 9-11 January 1991, Miami, Florida.

[15] 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.

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