An Introduction to Formal Hardware Verification

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

e-mail: seger@cs.ubc.ca

Abstract

Formal hardware verification has recently attracted considerable interest. The need for "correct" designs in safety-critical applications, coupled with the major cost associated with products delivered late, are two of the main factors behind this. In addition, as the complexity of the designs increase, an ever smaller percentage of the possible behaviors of the designs will be simulated. Hence, the confidence in the designs obtained by simulation is rapidly diminishing. This paper provides an introduction to the topic by describing three of the main approaches to formal hardware verification: theorem-proving, model checking, and symbolic simulation. We outline the underlying theory behind each approach, we illustrate the approaches by applying them to simple examples, and we discuss their strengths and weaknesses. We conclude the paper by describing current on-going work on combining the approaches to achieve multi-level verification approaches.

1 Introduction

Design validation involves taking steps to guarantee that a design will perform according to its specification. There are at least three levels of validation:

1. Design validation—have we designed what we intended to design?
2. Implementation validation—have we actually implemented our design?
3. Manufacturing validation[1]—have we actually manufactured our implementation?

In this paper we will concern ourselves only with design validation. Of course, we can never hope to verify that we have met our intentions, since intentions are, at best, vague and imprecise and not something we can write down and reason about (in any formal mathematical sense anyway). However, what we would like to achieve is that we can verify that our design has some verifiable properties.

[1]Traditionally called testing.

Simulation is often used as the main tool for "checking" a design. Typically a design team 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 simulated for an extended period of time. For example, if the design is a microprocessor, a design team might run some reasonably large programs (e.g. boot UNIX) on the simulated design. It is not uncommon to spend months of CPU time on mainframe computers simulating the final design[25].

Considerable effort has been made to simply increase, in a brute-force manner, what coverage can be achieved with simulation and to cut down the time it takes to achieve this coverage. One approach is to run a number of independent simulations distributing simulation cases over a set of machines. Another brute-force approach has been the development of special-purpose simulation hardware to increase the speed of a simulation by several orders of magnitude. However, despite these tremendous efforts, serious design errors often remain undetected.

To illustrate how hopeless traditional simulation is in checking a design, consider a 256 bit RAM[2]. Suppose we wanted to fully exhaustively simulate all possible state transitions of this circuit, how long would that take? A 256 bit RAM has approximately 10^{80} possible combinations of initial state and input. If we make the (very unrealistic) assumptions that we can use all matter in our galaxy to build computers (10^{17} kg), that each computer is of the size of single electron (10^{-30} kg), that each computer simulates 10^{12} cases per second and that we started at the time of the Big Bang (about 10^{10} years ago), we would just have reached the 0.05% mark of completing our task! Clearly, brute force approaches are doomed to fail.

However, even clever use of simulation is getting more difficult. In particular, 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. For example, in a heavily pipelined machine it is often impossible to simulate all possible instruction sequences that might lead to a trap. The difficulty here is that instructions that are logically unrelated all of a sudden are being processed in parallel and may even be processed out of order. Thus, the "quality" of the validation achieved by traditional simulation is rapidly deteriorating as the VLSI technology progresses.

Formal hardware verification attempts to overcome the weakness of non-exhaustive simulation by *proving* the correspondence between some abstract specification and the design at hand. In the long run, these formal approaches to hardware validation are better able to scale with the complexity of VLSI designs. The main reason for this is that they exploit powerful tools of mathematics rather than brute-force. A good example of this is the use of mathematical induction, which is a mainstay of the theorem-proving approach to formal hardware verification.

Before we continue, there is one question that needs to be answered. In the 1960's and early 1970's many had very high expectations for "software verification". However, gradually, these hopes fizzled out by the late 1970's. So why is hardware verification different? In our opinion, there are at least five major differences between hardware verification and software verification:

[2] The example is taken from [13].

1. Hardware designers, as opposed to software designers, are much more willing to use a very small set of primitives and obey some very restrictive rules on how to use them. As a consequence, hardware is often very regular and hierarchical.

2. Re-use of designs is common practice in hardware designs. Hence, much of the verification effort can be amortized over several products.

3. Hardware designers are already familiar and comfortable with writing detailed (although somewhat informal) specifications. For example, consider the wide-spread use of behavioral Verilog and VHDL models.

4. The primitives are significantly simpler. For example, the behavior of a latch is generally much easier to describe than the semantics of an assignment statement in a (realistic) programming language.

5. The cost of fixing bugs. In hardware designs, a design error can mean a six months delay. In todays highly competitive market, such a delay can mean the difference between success and failure.

The goal of this paper is to provide an introduction to practical modern formal hardware verification approaches. We have chosen three of the main approaches to formal hardware verification: theorem proving, model checking, and symbolic simulation. Each of these approaches are supported by software tools—many of which have been under constant development for the last decade—and each approach has different strengths and weaknesses. We briefly discuss the underlying theory behind the systems, illustrate their use, and discuss their different strengths and weaknesses. It should be emphasized that this paper is not a general survey paper covering all suggested approaches. For such a paper, the reader is referred to Yoeli's book[30] or Gupta's survey[22]. However, by describing three of the main approaches in some detail, we hope the interested reader will be well equipped to read the more general, and more terse, survey papers.

2 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, in the logic, proving that the specification and implementation were suitably related.

2.1 Underlying Theory

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[20][3]. 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 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 surprisingly small number of axioms. In the formulation used in the HOL system, only 5 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 quite complex ones, 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[4].

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 8 primitive rules of inference. The following three rules are representative examples:[5],

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

which allows us to make assumptions,

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

[3] We will only give the flavour of the theory. The interested reader is referred to [20] for the complete formulation.

[4] 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.

[5] 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 allows us to discharge an assumption, and

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

which is Modus Ponens.

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.

2.2 A Small Example

Now given the availability of mechanized theorem provers, how can they be used in formal hardware verification? The basic idea is to embed both the implementation and the specification in the formal logic used in the proof system. The correctness statements, like that every behavior of the implementation satisfies the specification, are then cast in terms of proving some relation in the chosen logic. To illustrate this process, we will use the HOL system as our theorem-prover and formulate our correctness results, model our implementation, and express our specification, in higher-order logic. The example we will use is trivial, but it illustrates many of the underlying ideas. It should be emphasized that the example is not representative for what can actually be achieved by using this approach.

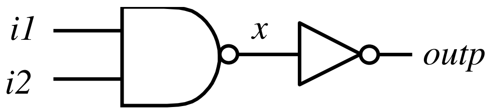

Figure 1: Implementation of two-input AND gate.

The task at hand is to show that *assuming* the NAND gate and the NOT gate behave as specified, then combining them as in Fig. 1 yields a two-input AND gate. In order to achieve this, we need to carry out four steps:

1. Specify the implementation of the AND gate.
2. Specify the behavioral models for the NAND and NOT gates.
3. Specify the intended behavior of the AND gate.
4. Prove that the implementation satisfies its intended behavior.

There are several ways to specify the implementation of the AND gate in higher-order logic. The most common way of doing this is using existential quantification to "hide" the internal connections, i.e., we would get:

$$\vdash_{def} \text{ANDgate\_IMP}(i1, i2, o) \equiv \exists x.\text{NANDgate}(i1, i2, x) \wedge \text{NOTgate}(x, o).$$

The behavioral model of the NAND and NOT gates can also be done in many ways in higher-order logic. Furthermore, different behavioral models can be used depending on the amounts of details needed or desired. Here, we will use a simple zero-delay model of their behavior. Hence,

$$\vdash_{def} \text{NANDgate}(i1, i2, o) \equiv o = \neg(i1 \wedge i2),$$

and

$$\vdash_{def} \text{NOTgate}(i, o) \equiv o = \neg i.$$

In a similar way, the desired behavior of the AND gate can be written as

$$\vdash_{def} \text{ANDgate\_SPEC}(i1, i2, o) \equiv o = i1 \wedge i2.$$

We are now faced with the task of formally proving that the implementation satisfies the specification. Before we do this, however, we need to define what it means for an implementation to satisfy some specification. Again, there are several ways of expressing this. For example, we could choose to verify that the input/output behavior of the implementation always agrees with the input/output behavior of the specification. In other words, we would have to verify that

$$\text{ANDgate\_IMP}(i1, i2, o) \equiv \text{ANDgate\_SPEC}(i1, i2, o)$$

holds. In other cases, e.g. when the specification is incomplete, we might be satisfied with verifying that the behavior of the implementation implies the behavior of the specification, i.e., that

$$\text{ANDgate\_IMP}(i1, i2, o) \Rightarrow \text{ANDgate\_SPEC}(i1, i2, o).$$

Suppose we choose the second type of correctness result, i.e., we want to verify that

$$\text{ANDgate\_IMP}(i1, i2, o) \Rightarrow \text{ANDgate\_SPEC}(i1, i2, o)$$

is a valid theorem. A "hand proof" of this result might look like:

1.	$ANDgate\_IMP(i1, i2, o)$	[assume initial formula]
2.	$\exists x.\text{NANDgate}(i1, i2, x) \wedge \text{NOTgate}(x, o)$	[by def. of ANDgate_IMP]
3.	$\text{NANDgate}(i1, i2, x) \wedge \text{NOTgate}(x, o)$	[strip off $\exists x$.]
4.	$\text{NANDgate}(i1, i2, x)$	[left conjunct of line 3]
5.	$x = \neg(i1 \wedge i2)$	[by def. of NANDgate]
6.	$\text{NOTgate}(x, o)$	[right conjunct of line 3]
7.	$o = \neg x$	[by def. of NOTgate]
8.	$o = \neg(\neg(i1 \wedge i2))$	[substitution, line 5 into 7]
9.	$o = (i1 \wedge i2)$	[simplify using $\neg(\neg(t)) = t$]
10.	$\text{ANDgate\_SPEC}(i1, i2, o)$	[by def. of ANDgate_SPEC]
11.	$\text{ANDgate\_IMP}(i1, i2, o) \Rightarrow \text{ANDgate\_SPEC}(i1, i2, o)$	[discharge assumption line 1]

Although, the above manual proof may appear tedious, it is still much shorter than the complete formal proof. For example, the HOL theorem prover establishes the result in 365 basic proof steps! Fortunately, the user only needs to guide the HOL theorem prover 4 times. This ratio of machine generated proof steps and manual interactions by a user guiding the theorem prover of close to 100 is relatively low for interactive theorem provers. Many systems can often achieve a factor closer to 1000[6].

The above example was of course very simple. In particular, the models of a NAND and NOT gate were overly simplified. However, since we have the full expressive power of logic at our disposal, it is quite simple to generalize the behavioral models for the individual components. In this way delays and delay models, for example, can be introduced. Of course, the more complex the behavioral model is, the more complex the correctness proof will be.

2.3 Strengths and Weaknesses

The main advantage of a formal proof is that it can be mechanically checked. The main disadvantage, compared to ordinary human reasoning, is that deriving a formal proof is overwhelmingly tedious. Fortunately, there has been considerable progress made towards the partial automation of formal proof. A very large fraction of the actual line-by-line inference steps in a formal proof can be generated automatically by a computer-based theorem-prover.

A digital circuit can be "verified" using a theorem-prover by generating a theorem which states that the formal specification of a design logically *satisfies* a formal specification of its intended behaviour (i.e. a high level model). The exact meaning of "satisfies" is stated unambiguously as a mathematical relationship between the two levels of formal specification. This can be judiciously interpreted as a guarantee that the design is correct, with respect to the high level model, for all possible cases.

Among the best known interactive theorem-provers are the Boyer-Moore Theorem Prover [6] and the Cambridge HOL System [21]. The Boyer-Moore Theorem Prover has been used by researchers at Computational Logic Inc. to develop an 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 [16].

A distinctive feature of the theorem-proving approach is that it is structural rather than behavioural. Consequently, a 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. There are both pros and cons to this approach. Clearly, using hierarchy is a very good way of managing complexity. Unfortunately, circuits do not always have nice hierarchical structure. This is especially true after aggressive optimization have been performed.

[6] Despite this quite large multiplicative effect, we believe this number must be pushed closer to 10,000 or even 100,000 before interactive theorem provers will be straightforward to use for non-experts. As it is today, most theorem provers require quite sophisticated users to be effective.

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 Model Checking

One of the characteristics of the theorem-proving approach is its structural rather than behavioural view of the verification process. Model checking takes the completely opposite approach. Here only the behavior of some system is checked and verified to satisfy some properties. In general, model checking is an algorithm that can be used to determine the validity of formulas written in some temporal logic with respect to a behavioral model of a system. For example, if the controller of a microprocessor is described as a state machine, a model checker can be used to verify that only the trap instruction can change the mode of the processor from user mode to supervisor mode. It can also be used to prove properties like that an interrupt is acknowledged at most t time units after the interrupt request, etc.

3.1 Underlying Theory

Before going into the details of the model checking algorithm, we need to introduce temporal logic. Propositional logic deals with absolute truths in a domain, i.e., given a domain, propositions are either true or false. Predicate logic extends this notion of truth to make it relative, in that the truth of a predicate depends on the actual arguments involved. Extending this notion even further, in modal logic, of which temporal logic is a special case, the truth of a formula may depend on what "world" we are currently in. Within a world, predicate logic is used, between worlds, modal operators are introduced. In the hardware domain (as well as in most other practical systems) the different worlds represent different states of the system and the movement from one world to another represent the dynamic behavior of the system. In this paper we will consistently use the word state, rather than world. In temporal logic, the fact that a state t is a successor state of s implies that t is a possible next state of the system. There is a wealth of temporal logics[7]. In this section we will highlight only one—computational tree logic (CTL)—together with its associated decision procedure[19].

[7] For a very comprehensive discussion of the different temporal logics used in hardware verification related work, the reader is referred to [22].

A CTL formula is defined with respect to some set of atomic formulas. These atomic formulas should be thought of as basic properties of the individual states. In a hardware context, they could include statements like "gate output x is high", "gate output y is low", etc. Formally, the syntax of a CTL formula is defined as follows[8]:

1. Every atomic formula is a CTL formula.
2. If f is a CTL formula, then so are
   (a) $\neg f$ (not f),
   (b) AXf (for all paths f holds in the next state),
   (c) EXf (there is a path in which f holds in the next state),
   (d) AGf (for all paths, f holds in every state),
   (e) EGf (for some path, f holds in every state),
   (f) AFf (for all paths, eventually f holds),
   (g) EFf (for some path, eventually f holds),
3. If f and g are CTL formulas, then so are:
   (a) $f \wedge g$ (f and g),
   (b) $A(fUg)$ (for all paths: f until g),
   (c) $E(fUg)$ (for some path: f until g).

All the remaining logical connectives like $\vee$, $\Rightarrow$, etc., are defined in the usual way in terms of $\wedge$ and $\neg$.

The semantics of a CTL formula is defined with respect to a labeled, directed graph, called a CTL structure[9]. Formally, a CTL structure is a quadruple $M = (S, R, A, L)$, where

1. S is a finite set of states.
2. R is a binary relation on S. sRt means that t is a possible immediate successor of s. We assume that R is total, i.e., every state has at least one successor state.
3. A is a set of atomic formulas.
4. $L : A \longrightarrow 2^S$ is a function that maps each atomic formula into the set of states in which the formula holds.

Let now s be a state in the CTL structure $M = (S, R, A, L)$. With M and s we associate an infinite computation tree, rooted at s and with an arc from node t to node u iff tRu. Given a CTL formula f, we write $M, s \models f$ (or $s \models f$ if M is fixed), to state that the formula f holds in the computation tree derived from M and rooted at s. An infinite path of the tree, starting at the root, is an s-path of M.

Formally, given the CTL structure $M = (S, R, A, L)$, the semantics of a CTL formula f is defined recursively as:

[8] We include in parenthesis the common way to read the different symbols.
[9] One can view this structure as a finite Kripke structure[23].

1. $s \models a$ iff $a \in A$ and $s \in L(a)$, i.e., a is an atomic formula of M and a holds in s.

2. (a) $s \models \neg f$ iff $s \models f$ does not hold,
   (b) $s \models AXf$ iff $t \models f$ for every t such that sRt.
   (c) $s \models EXf$ iff $t \models f$ for some t such that sRt.
   (d) $s \models AGf$ iff f holds in every state in every s-path of M.
   (e) $s \models EGf$ iff f holds in every state in some s-path of M.
   (f) $s \models AFf$ iff for every s-path of M there is at least one state in which f holds.
   (g) $s \models EFf$ iff for some s-path of M there is at least one state in which f holds.

3. (a) $s \models f \wedge g$ iff $s \models f$ and $s \models g$.
   (b) $s \models A(fUg)$ iff for every s-path $(s_0, s_1, s_2, \ldots)$ of M there exists some $j \geq 0$ such that $s_j \models g$ and $s_i \models f$ for $0 \leq i < j$.
   (c) $s \models E(fUg)$ iff for some s-path $(s_0, s_1, s_2, \ldots)$ of M there exists some $j \geq 0$ such that $s_j \models g$ and $s_i \models f$ for $0 \leq i < j$.

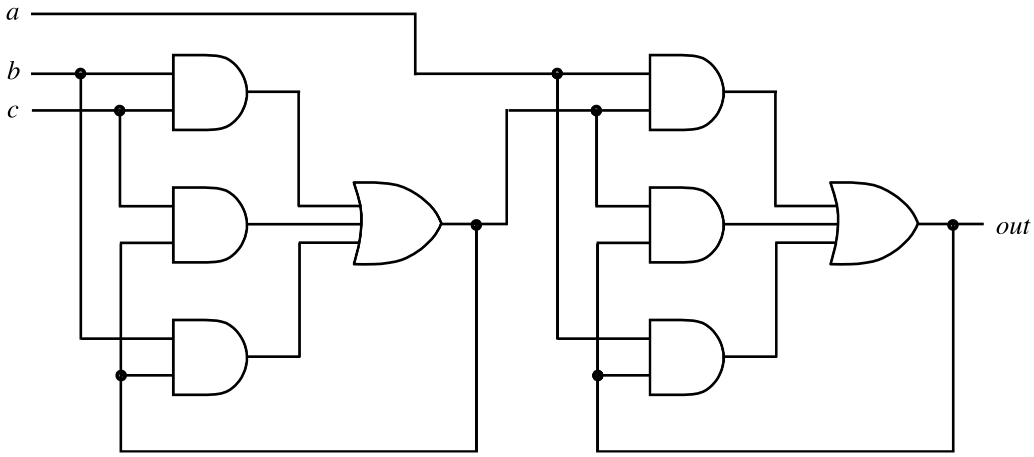

Figure 2: Implementation of three-input C-element.

The temporal logic described above is quite useful for describing desirable properties of a design. To illustrate this, consider the gate circuit shown in Fig. 2. The circuit is meant to implement a three input Muller C-element. We can divide the properties we would like the circuit to exhibit into two types: *liveness* (something good will eventually happen) and *safety* (nothing bad will ever happen). First the liveness properties. For a C-element to function properly, we should make sure that if we keep all the inputs equal, then eventually the output should change to this value. We can express this property in the following two CTL formulas.

$$AG(A((a = 0 \wedge b = 0 \wedge c = 0)\, U\, (out = 0 \vee a = 1 \vee b = 1 \vee c = 1)))$$

$$AG(A((a = 1 \wedge b = 1 \wedge c = 1)\, U\, (out = 1 \vee a = 0 \vee b = 0 \vee c = 0)))$$

Basically, the CTL formulas state that either the inputs must change or the output must change to the proper value.

A safety condition that should hold is of course that if all the inputs have same value and the output also has this value, then the output should not change until all the inputs have changed to their complementary values. This condition can be expressed in the two CTL formulas:

$$AG((a = 0 \wedge b = 0 \wedge c = 0 \wedge out = 0) \Rightarrow A(out = 0\, U\, (a = 1 \wedge b = 1 \wedge c = 1)))$$

and

$$AG((a = 1 \wedge b = 1 \wedge c = 1 \wedge out = 1) \Rightarrow A(out = 1\, U\, (a = 0 \wedge b = 0 \wedge c = 0)))$$

Although, CTL is a concise and powerful specification language to describe desirable properties of a system, this is only one of the reasons for our interest in CTL. The second reason is that there is a very efficient algorithm for determining whether a CTL formula holds in a given state in a CTL structure. The basic algorithm, called the model checking algorithm[19], was developed by E. Clarke, E. Emerson, and A. Sistla. The original algorithm was described in terms of labeling the CTL structure. Unfortunately, this requires an explicit representation of the whole state space. Here we will present the algorithm in terms of fixed point calculations. The main reason for this is that the algorithm is then amenable to symbolic formulation. Symbolic evaluation allow us to determine the validity of a CTL formula without having to explicitly enumerate all states of the CTL structure—a significant improvement of the maximum system we can deal with.

Before we present the algorithm, we need to introduce the basic idea of fixed point calculations. For a more complete treatment of the underlying theory, the reader is referred to [29]. Given a set S, the power set of S, denoted 2^S, is the set of all subsets of S. We can introduce a complete partial order $\sqsubseteq$ on 2^S defined as: $a \sqsubseteq b$ iff $a \subseteq b$. It is easy to verify that this partially ordered set forms a complete lattice. Hence, the least upper bound and the greatest lower bound are defined[10] for any subset of 2^S. Now given a function $h : 2^S \longrightarrow 2^S$, we say that the function is continuous iff for every increasing sequence $x_1 \sqsubseteq x_2 \sqsubseteq \ldots$, where $x_i \in 2^S$, we have $h(\sqcup_{i\geq 1} x_i) = \sqcup_{i\geq 1} f(x_i)$ and $h(\sqcap_{i\geq 1} x_i) = \sqcap_{i\geq 1} f(x_i)$, where $\sqcup$ and $\sqcap$ denotes the least upper bounds and greatest lower bounds respectively. Now given a continuous function $h : 2^S \longrightarrow 2^S$ one can prove that:

1. (a) h has a least fixed point $b \in 2^S$, i.e., b satisfies $h(b) = b$ and if $h(c) = c$ for some $c \in 2^S$ then $b \sqsubseteq c$.
   (b) The least fixed point of the function h, written *Lfp* $U.h(U)$, is defined by $\sqcup_{i\geq 0} h^i(\emptyset)$, where h^i is the composition of i copies of h.
2. (a) h has a greatest fixed point $d \in 2^S$, i.e., d satisfies $h(d) = d$ and if $h(c) = c$ for some $c \in 2^S$ then $c \sqsubseteq d$.
   (b) The greatest fixed point of the function h, written *Gfp* $U.h(U)$, is defined by $\sqcap_{i\geq 0} h^i(S)$.

The fixed point calculations are used to compute the set of states that satisfies some of the "global" CTL formulas. For example, consider finding the set of states that satisfies the CTL formula AGf when given the set $H(f)$ that contains all the states that satisfies f. Below we will state that this can be accomplished by computing the greatest fixed point of the function $h(U) = H(f) \cap \{s \mid \forall t\; sRt \Rightarrow$

[10] They correspond to set union and set intersection respectively.

$t \in U\}$[11]. Intuitively, what we are doing here is finding the largest subset U of S such that 1) f holds in every state in U and 2) for every element $u \in U$, every successor state of u is also in U. Clearly, every element $u \in U$ will satisfy AGf. Conversely, it is easy to convince oneself that if an element v is not in U, then either f does not hold in v or there is some state reachable from v in which f does not hold (otherwise v would have been in U). However, that means immediately that AGf does not hold in v. Altogether, it is fairly straightforward to prove that $s \models AGf$ iff $s \in \mathit{Gfp}\ U.\ H(f) \cap \{s \mid \forall t\ sRt \Rightarrow t \in U\}$.

Given a CTL structure $M = (S, R, A, L)$, and a CTL formula f, the model checking algorithm computes the set of states $H(f) \subseteq S$ that satisfies f. The algorithm is defined recursively as follows:

1. $H(A) = \{s \mid A \in L(s)\}$.
2. (a) $H(\neg f) = S - H(f)$
   (b) $H(AXf) = \{s \mid \forall t\ sRt \Rightarrow t \in H(f)\}$.
   (c) $H(EXf) = H(\neg(AX(\neg f)))$
   (d) $H(AGf) = \mathit{Gfp}\ U.\ H(f) \cap \{s \mid \forall t\ sRt \Rightarrow t \in U\}$.
   (e) $H(EGf) = H(\neg(AF\neg f))$.
   (f) $H(AFf) = \mathit{Lfp}\ U.\ H(f) \cup \{s \mid \forall t\ sRt \Rightarrow t \in U\}$.
   (g) $H(EFf) = H(\neg(AG\neg f))$.
3. (a) $H(f \wedge g) = H(f) \cap H(g)$.
   (b) $H(A(fUg)) = \mathit{Lfp}\ U.\ H(g) \cup (H(f) \cap \{s \mid \forall t\ sRt \Rightarrow t \in U\}$.
   (c) $H(E(fUg)) = H(\neg(A(\neg gU(\neg f \wedge \neg g))\ \vee\ AG(\neg g)))$.

3.2 A Small Example

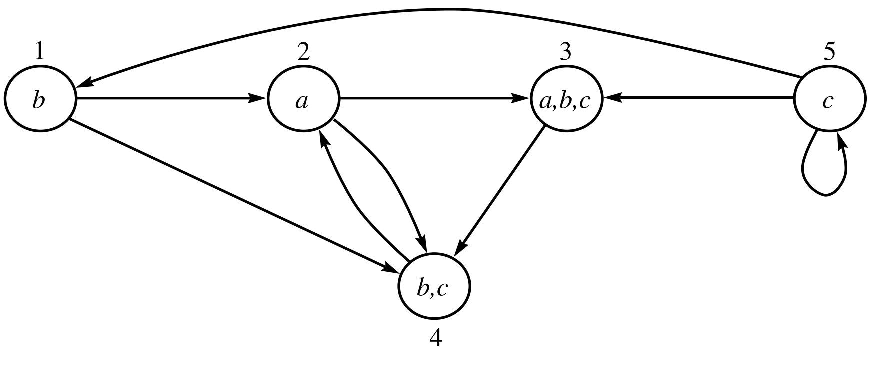

Figure 3: Simple CTL structure.

To illustrate the model checking algorithm, consider the CTL structure $M = (S, R, A, L)$, where $S = \{1, 2, 3, 4, 5\}$, $A = \{a, b, c\}$ and R and L can be obtained from Fig. 3. Now consider the CTL

[11] We leave it as an exercise to the reader to verify that h is indeed continuous and thus the greatest fixed point is well defined.

formula $AG(a \vee c)$. We would like to compute the set of states of S in which this CTL formula holds. In other words, we would like to find the set of states U, such that:

1. every state in U is labeled with a or c (or both).
2. for every state $u \in U$ and every infinite path starting in u, every state in this path is labeled with a or c.

First, we rewrite the formula by using the usual logical connectives, i.e., we rewrite $AG(a \vee c)$ as $AG(\neg((\neg a) \wedge (\neg c)))$. Now, using the definition of $H()$ we obtain:

$$H(AG(\neg((\neg a) \wedge (\neg c)))) = \mathit{Gfp}\ U.\ H(\neg((\neg a) \wedge (\neg c))) \cap \{s \mid \forall t\ sRt \Rightarrow t \in U\}.$$

Now, since

$$\begin{aligned} H(\neg((\neg a) \wedge (\neg c))) &= S - H((\neg a) \wedge (\neg c)) \\ &= S - (H((\neg a)) \cap H((\neg c))) \\ &= S - ((S - H(a)) \cap (S - H(c))) \\ &= S - ((S - \{2,3\}) \cap (S - \{3,4,5\})) \\ &= S - ((\{1,4,5\}) \cap (\{1,2\})) \\ &= \{2,3,4,5\} \end{aligned}$$

we get

$$H(AG(\neg((\neg a) \wedge (\neg c)))) = \mathit{Gfp}\ U.\ \{2,3,4,5\} \cap \{s \mid \forall t\ sRt \Rightarrow t \in U\}.$$

The fixed point can now be computed as:

$$\begin{aligned} U_0 &= S \\ \\ U_1 &= \{2,3,4,5\} \cap \{s \mid \forall t\ sRt \Rightarrow t \in U_0\} \\ &= \{2,3,4,5\} \cap \{1,2,3,4,5\} = \{2,3,4,5\} \\ \\ U_2 &= \{2,3,4,5\} \cap \{s \mid \forall t\ sRt \Rightarrow t \in U_1\} \\ &= \{2,3,4,5\} \cap \{1,2,3,4\} = \{2,3,4\} \\ \\ U_3 &= \{2,3,4,5\} \cap \{s \mid \forall t\ sRt \Rightarrow t \in U_2\} \\ &= \{2,3,4,5\} \cap \{1,2,3,4\} = \{2,3,4\} = U_2 \end{aligned}$$

and thus

$$\begin{aligned} H(AG(\neg((\neg a) \wedge (\neg c)))) &= \mathit{Gfp}\ U.\ \{2,3,4,5\} \cap \{s \mid \forall t\ sRt \Rightarrow t \in U\} \\ &= \{2,3,4\}. \end{aligned}$$

Altogether, we can conclude that the CTL formula $AG(a \vee c)$ holds in the states 2, 3, and 4.

3.3 Strengths and Weaknesses

Since the dynamic behavior of a hardware system is usually the most difficult to design and verify, temporal logic is a very precise and (mostly) convenient formalism for expressing desired properties of the system. In particular, the branching time temporal logic CTL, discussed above, is very convenient for reasoning about asynchronous and self-timed designs.

The main strength of many of these temporal logics, and CTL in particular, is the fact that the decision procedure is completely automated. Thus, the user never needs to be aware of the CTL structure, but can simply interact with the model checker to determine that the dynamic behavior of the design satisfies the specification. An interesting observation is that the CTL structure is very general. Hence, the same model checker can be used to reason about both hardware and software systems.

There are two major drawbacks with temporal logics and model checking. First, in this approach to hardware verification, the specification is as an enumeration of desired properties. However, it is often quite difficult to judge whether the temporal formulas that has been checked completely characterizes the desired behavior of the system. For example, it is easy to forget to check some property that one might take for granted[12]. Also, for a practical point of view, the temporal logic formulas can sometimes become exceedingly difficult to understand. Hence, there is a danger of misunderstanding what properties *actually* have been verified.

The second major drawback with temporal logic is related to its decision procedure. In order to determine the validity of a temporal logic formula it is necessary to extract the CTL structure for the device. The problem is that the state space S for most "interesting" systems is extremely large. In particular, if the system consists of many asynchronous communicating state machines, the complete state space is often enormous. This problem is usually referred to as the "state explosion problem". The topic of how to deal with the state explosion problem is currently a very active area of research. The most promising approach is symbolic methods[14]. In this approach the explicit construction of the state graph is avoided. Instead, the state graph is represented by means of functions from sets of states to sets of states. In particular, the sets are represented by their characteristic functions over Boolean variables. By using these techniques, system with more than 10^{20} states have been successfully checked.

Although symbolic model checking has significantly increased the useful domain for the method, the achievable state space is still much too small to model circuits that include non-trivial data paths. Hence, model checking is primarily useful in verifying the control parts of a design. Other methods must be used to verify the datapath as well as verifying the interactions between the datapath and the control parts.

[12] Of course, one can also take the view that the fact that expressing the specification in temporal logic—which is behavioral rather than structural and thus will be completely different from the design—makes it less likely to make the same mistake in both the design as well as in the specification.

4 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 researcher at IBM Yorktown Heights in the late 1970's as a method for evaluating register transfer language representations [15]. 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 [10] 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 [9]. 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.

It is important to realize that if a circuit passes an "exhaustive" simulation suite, that does not necessarily mean that the circuit is formally verified correct. In order to prove a correctness result, not only must all possible input values be simulated, all possible initial circuit states must also be taken into account. Hence, a verification strategy is needed as well as a sophisticated symbolic simulator.

Recently, Bryant and Seger [12] 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. Further, by exploiting the 3-valued modeling capability of the simulator, where the third logic value X indicates an unknown or indeterminate value, the complexity of the symbolic manipulations is reduced considerably.

This verifier supports a verification methodology in which the desired behavior of the circuit is specified in terms of a set of assertions, each describing how a circuit operation modifies some component of the (finite) state or output. The temporal logic allows the user to define such interface details as the clocking methodology and the timing of input and output signals. The combination of timing and state transition information is expressed by an assertion over state trajectories giving properties the circuit state and output should obey at certain times whenever the state and inputs obey some constraints at earlier times.

In addition to this abstract behavior specification, the user is also required to describe how the circuit realizes the abstract system state. This mapping is given as an encoding of the abstract state in terms of binary values on circuit nodes at different times in the clock cycle.

This form of specification works well for circuits that are normally viewed as *state transformation systems*, i.e., where each operation is viewed as updating the circuit state. Examples of such systems include memories, data paths and processors. For such systems, the complex analysis permitted by model checkers is not required.

4.1 Underlying Theory

In this subsection we will highlight the underlying theory for symbolic trajectory evaluation. Although the general theory is equally applicable to hardware as software systems, in this paper we will describe a somewhat specialized version tailored specifically to hardware verification. For the more general theory, the reader is referred to [26] and [12].

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. Formally, this is expressed by defining an *assignment* ϕ to be a particular mapping from the elements of $\mathcal{V}$ to binary values. A formula F in the logic expresses some property of the circuit in terms of the symbolic variables. It may hold for only a subset $\mathcal{D}$ of the possible assignments. Such a subset can be represented as a Boolean domain function d over $\mathcal{V}$ yielding 1 for precisely the assignments in $\mathcal{D}$. The constant functions $\mathbf{0}$ and $\mathbf{1}$, for example, represent the empty and universal sets, respectively.

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 extremely limited. A formula in this logic is either:

1. *UNC* (unconstrained),
2. (a) $\mathtt{n}=1$ (a node is equal to 1),
   (b) $\mathtt{n}=0$ (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 following state).

The temporal logic supported by the evaluator is far weaker than that of other 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 *OK* is computed expressing those assignments for which the assertion holds. For a correct circuit, this function should equal $\mathbf{1}$; otherwise, the negation of the function provides counterexamples.

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 generates automatically the assertions and carries out the verification process.

The Voss system is a formal verification system based on symbolic evaluation of partially-ordered trajectories developed at University of British Columbia. Conceptually, the Voss system consists of two parts. A "simulator" back-end and a "language" front-end, see Figure 4. The front-end is a compiler/interpreter for a small, fully lazy, functional language with a precisely defined semantics. A specification is written as a "program" in this language. Of course, since the language is fully functional, one can also view a program simply as defining a (complex) function. When this specification program is executed, it builds up the simulation sequence that must be run in order to completely verify the specification.

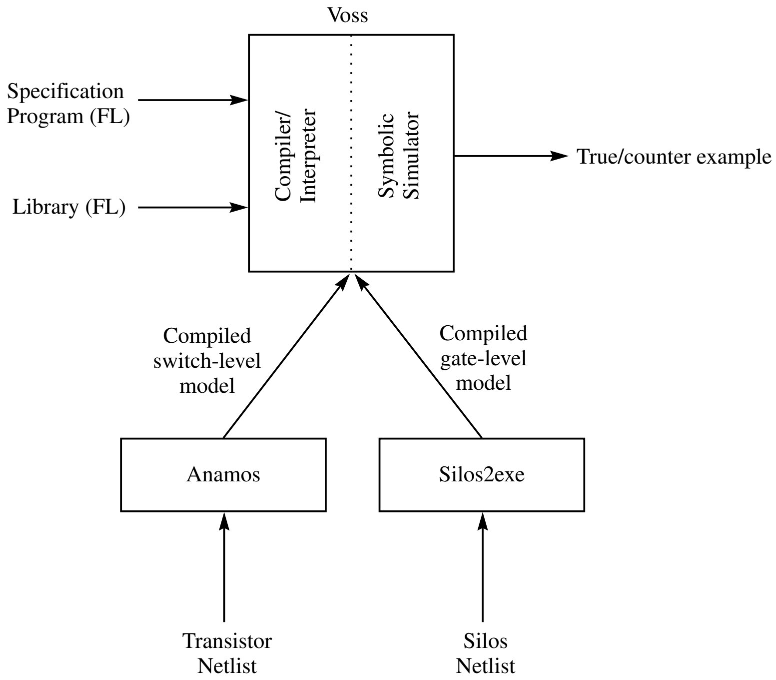

Figure 4: Voss verification system

The back-end is an extended symbolic switch-level simulator. The simulator can handle both switch- and gate-level descriptions. The switch-level simulator is an offspring of the COSMOS[11] switch-level simulator and thus supports the MOSSIMII [8] switch-level model. The gate level simulator is (roughly) functionally equivalent to the SILOS II [28] simulator. In addition, more comprehensive delay modeling capabilities has been added for more accurate verification. The simulator employs event-scheduling during both the circuit simulation as well as in maintaining the verification conditions in order to achieve good performance.

4.2 A Small Example

To illustrate the use of the Voss system, consider the circuit shown in Fig. 5. This is a 16-bit instance of a (pseudo) domino-logic design for a circuit that tests whether: 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. Note that this circuit is by no means representative for the size and complexity of circuits that the approach capable of handling. (For example, the circuit is (essentially) a combinational circuit whereas virtually all "interesting" designs are sequential.) However, as an example it illustrates several points.

A specification expressed in the Voss system language is given below for the circuit in Fig. 5.

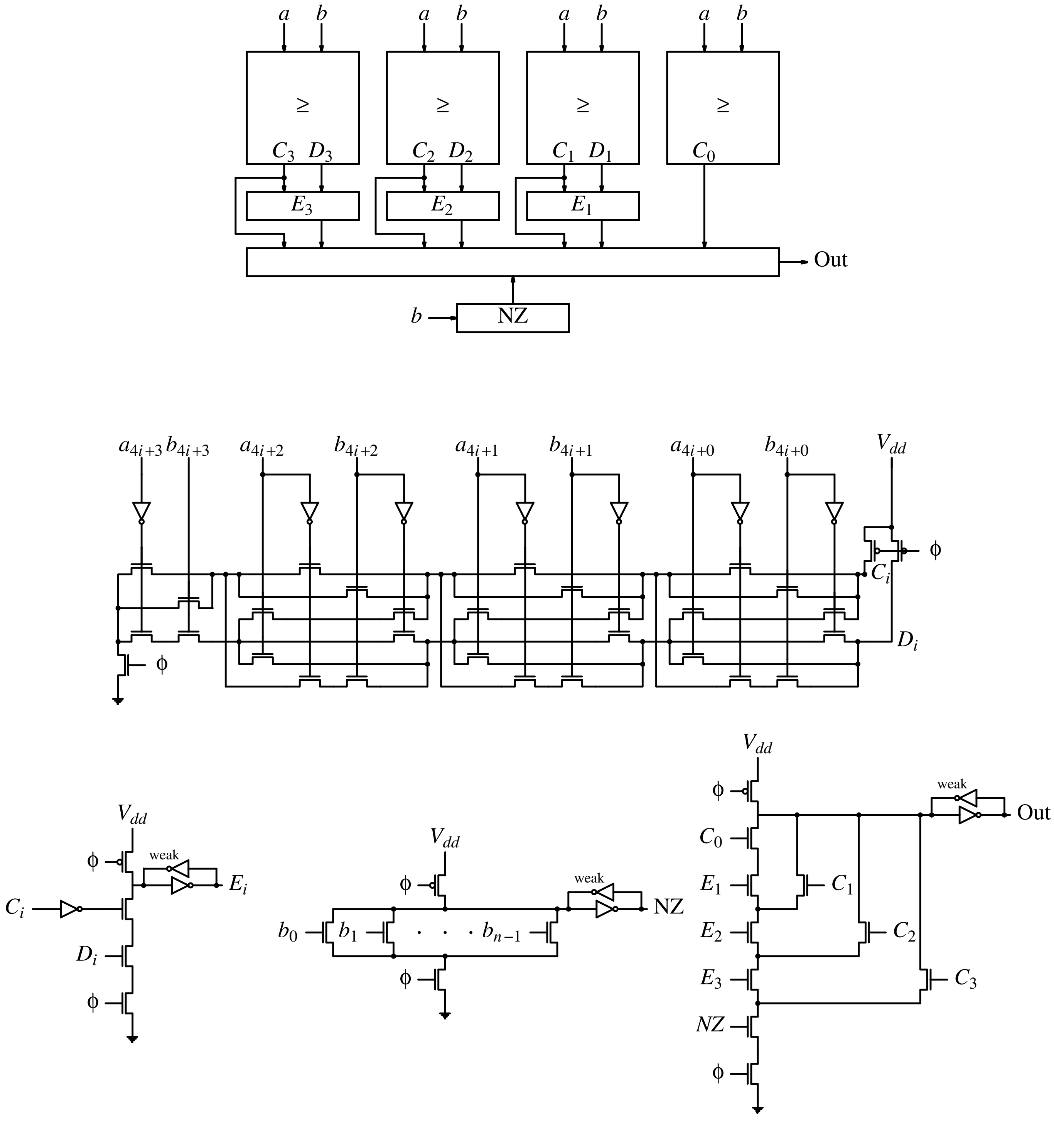

Figure 5: 16-bit circuit for computing $a > b > 0$.

```
load "verification.fl"; // Load library routines
let N = 16;             // Size of circuit

// Nodes accessed
let phi = node "phi1";
let out = node "out";
let an  = node_vector "a." N;
let bn  = node_vector "b." N;

// Variables used
let av  = variable_vector "a" N;
let bv  = variable_vector "b" N;

// Functionality specification
letrec greater al bl = (empty al) => F |
        let eq a b = NOT (a XOR b) in
        let ai = (hd al) in
        let bi = (hd bl) in
        (ai AND (NOT bi)) OR ((ai _eq bi) AND ((tl al) _greater (tl bl)));

letrec not_zero al = (empty al) => F | ((hd al) OR (not_zero (tl al)));

let CMP_BitLevel av bv = (av _greater bv) AND (not_zero bv);

// Verification conditions
let Timing av bv out_val =
             verify (((phi _is F _for 100) _then (phi _is T _for 100)) _and
                     (((an _is av) _and (bn _is bv)) _from 95 _to 200))
//           implies
                    (out _is out_val  _from 180 _to 200);
```

The above definition of `CMP_BitLevel` specifies the bit level "compare operation". `Timing` describes the timing conditions under which we wish to verify the circuit in Fig. 5. To paraphrase this definition: on the assumption that,

- the clock signal `phi` is low for 100 time units and then is high for another 100 time units,
- the vectors of circuit nodes `a` and `b` are assigned the vectors of symbolic Boolean variables `av` and `bv` 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 `CMP_BitLevel` and the timing conditions expressed by `Timing` are combined in the top level Voss specification: `Timing av bv (CMP_BitLevel av bv)`

The Voss system can now be used to fully automatically derive a correctness theorem for the circuit of Fig. 1. More specifically, the Voss system can deduce that the above specification is a logical consequence of the finite state machine derived from the netlist and the built-in switch-level simulator for the circuit shown in Fig. 5. The CPU time required to run this verification task is less than a second on a SPARC II.

4.3 Strengths and Weaknesses

One key property of symbolic trajectory evaluation is that it involves symbolic manipulation only over those variables explicit in the assertion. In contrast, symbolic model checking algorithms must perform manipulations involving as many variables as there are bits of state in the circuit. This difference can be very significant.

Unlike the theorem-proving approach, symbolic trajectory evaluataion[13] does not require a hierarchy of behavioural 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 completely different implementations. Also, the exact details of how the circuit works can often be left to the simulator to compute. In particular, it 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 bits 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 evaluatation 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. For example, the Voss specification of the circuit shown in Fig. 5 is in terms of vectors of Boolean variables. However, in the mind of the designer, the intended behaviour should be stated in terms of an arithmetic relation—$out = 1$ iff $A > B > 0$. Since the specification used in the symbolic trajectory evaluator often consists of a (simple) implementation of the desired functionality, there is a danger that a designer might use an incorrect assumption in both deriving the specification as when designing the circuit itself.

5 Future of Formal HW Verification

As outlined earlier in Section 1, conventional techniques for design validation (based mainly on traditional simulation) cannot be expected to keep pace with the increasing complexity of VLSI designs. Formal techniques such as symbolic simulation, symbolic model checking, and theorem-proving are

[13]The same holds for model checking too.

better able to scale with this increasing complexity by exploiting powerful mathematical tools rather than brute-force. However, certain "concerns" have so far limited the practical application of these formal techniques outside the domain of research. We believe the two main concerns can be stated as:

- There is a wide "gulf" between conventional Computer-Aided Design approaches to design validation (e.g., traditional switch-level simulation) and formal verification techniques, especially general purpose theorem-proving. The adoption of formal techniques currently requires a revolutionary change to the design cycle.
- Each approach to formal hardware verification has its own strengths and weaknesses. It is extremely unlikely that only one approach can be used for all the verification needs of a designer. Unfortunately, todays verification systems make it virtually impossible to combine different approaches to a single verification task. In fact, even the format of describing circuits are often so incompatible that many tools need hand-translated descriptions as their inputs!

Both of these statements must be addressed if formal hardware verification is to gain wide acceptance in industry.

5.1 Multi-Level Verification Approaches

Traditionally, the formal hardware verification community has been divided into different camps with very little interactions between different camps. Clearly, this must change, since the different approaches often have complementary strengths. In this subsection we will illustrate this by comparing, and contrasting, the theorem proving approach with the symbolic trajectory evaluation approach. The material in this section is a summary of [27].

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

- in symbolic trajectory evaluation, the underlying model of hardware is built-in (and thus, is fixed); the user specifies particular designs on top of this fixed model.
- in theorem-proving, there is no built-in model of the underlying hardware; the user must supply the underlying model of the hardware before specifying particular designs.
- in symbolic simulation, because the underlying model of the hardware is built-in, the lowest level of specification is fixed.
- in theorem-proving, because the underlying model of the hardware is supplied by the user, the lowest level of specification is variable.
- 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 behaviour very efficiently—and the the results are indeed convincing. Hence, an ideal verification tool would draw from both methodologies.

To illustrate how such "two-level" verification tool could be used, consider again the circuit of Fig. 5. 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. 5 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. 5 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. In [27] is outlined how this proof could be carried out using a combined verification approach. The basic idea is to embed the Voss language in HOL, and thus establish a "mathematical interface" based on precisely defined semantics between the Voss system and HOL. Once this has been accomplished, the two tools can be used in tandem using Voss for lower-levels verification and HOL at higher levels.

5.2 Integration into the Design Cycle

It is often thought that the integration of formal verification techniques into industrial practice requires a revolutionary change in design methodology. However, in view of the on-going work on multi-level verification mentioned above, we believe that these techniques can in fact be integrated into industrial practice in an evolutionary rather than revolutionary manner. In particular, consider the following "capability maturity model" for formal hardware verification.

The entry point of this capability model would be the unsophisticated use of symbolic simulation as traditional simulation, i.e., not use the symbolic capabilities of the simulator. This is merely replacing existing simulators with another simulator[14]. The next level would involve symbolic representation of data which can result in more effective use of symbolic simulation. The next level of this capability model would be the use of "analytical" specifications of desired behaviour which better exploit the symbolic nature of this approach to formal hardware verification. In particular, symbolic trajectory evaluation using various forms of "symbolic indexing"[1] would now be used. Higher levels of this model could then be defined in terms of the incremental use of various theorem-proving capabilities.

In general, we do not believe formal methods will completely replace conventional simulation. However, there is reasons to believe that formal methods can start to replace the most time consuming simulations. Using the capability maturity model outlined above would gradually lead to this as the designer becomes more and more confident in the tool, the approach, and in his/her own capabilities

[14] Of course, for this replacement to be acceptable, the symbolic simulator must be at least as capable as the replaced simulator and be able to use the same simulation scripts as the existing simulator.

in using the formal verification tools.

Acknowledgements

Thanks are due to the members of the Integrated Systems Design group in the Computer Science Department of the University of British Columbia for providing a very exciting and stimulating environment for studying formal methods. In particular, special thanks go to Jeff Joyce for many (and lengthy) discussions on the strengths and weaknesses of theorem proving and symbolic trajectory evaluation. Also, Jeff's comments on an earlier draft of this paper resulted in a significantly improved paper. This work was supported by an operating grant from the Natural Sciences Research Council of Canada and by an Advanced Systems Institute Fellowship.

References

[1] 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.

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

[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] G.V. Bochmann, "Hardware Specification with Temporal Logic: An Example", *IEEE Transactions on Computers*, Vol. C-31, No. 3, March 1982, pp. 223-231.

[5] S. Bose and A.L. Fisher, "Automatic verification of Synchronous Circuits using Symbolic Logic Simulation and Temporal Logic", *IMEC IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, 1989, pp. 759-764.

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

[7] M. Browne, E. Clarke, D. Dill, and B. Mishra, "Automatic Verification of Sequential Circuits using Temporal Logic", *IEEE Transactions on Computers*, Vol. C-35, No. 12, December 1986, pp. 1035-1044.

[8] 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.

[9] R.E. Bryant, Symbolic Verification of MOS Circuits. *1985 Chapel Hill Conference on VLSI*, Fuchs, H., Ed. Computer Science Press, Rockville, MD, 1985, pp. 419–438.

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

[11] R. E. Bryant, *et al*, "COSMOS: A Compiled Simulator for MOS Circuits," *24th Design Automation Conference*, 1987, pp. 9–16.

[12] 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).

[13] R. E. Bryant, "Tutorial on Formal Verification of Hardware," *28th Design Automation Conference*, 1991.

[14] J. R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking", *27th Design Automation Conference*, ACM, New York, 1990.

[15] 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.

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

[17] O. Coudert, C. Berthet, and J.C. Madre, "Verification of Sequential Machines using Boolean Functional Vectors", *IMEC IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, 1989, pp. 111-128.

[18] O. Coudert, J.C. Madre, and C. Berthet, "Verifying Temporal Properties of Sequential Machines Without Building their State Diagram", *DIMACS Workshop on Computer-Aided Verification*, Rutgers, June 1990.

[19] E. M. Clarke, E. A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach", *10th ACM Symposium on Principles of Programming Languages*, ACM, New York, 1983.

[20] M. Gordon, *HOL: A Machine Oriented Formulation of Higher Order Logic*, Cambridge University, Computer Laboratory Technical report No. 68, Cambridge, England, 1985.

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

[22] A. Gupta, *Formal Hardware Verification Methods: A Survey*, Carnegie Mellon University, Technical report CMU-CS-91-193, Pittsburgh, PA, 1991.

[23] G. E. Hughes and M.J. Creswell, *An introduction to Modal Logic*, Metheuen and Co., 1977.

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

[27] C-J. Seger and J. Joyce, "A Two-Level Formal Verification Methodology using HOL and COSMOS", *Workshop on Computer-Aided Verification*, K. Larsen and A. Skou, eds., Aalborg University, Denmark, July 1991, pp. 380-391.

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

[29] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge Massachusetts, 1977.

[30] M. Yoeli, *Formal Verification of Hardware Designs*, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.