Formal Verification of Sequential Hardware: A Tutorial

Michael C. McFarland, *Member, IEEE*

Abstract—**Formal verification involves the use of analytical techniques to prove that the implementation of a system conforms to the specification. The specification could be a set of properties that the system must have, or it could be an alternative representation of the system behavior. Recently, there has been increased interest in the role of formal verification in hardware development. Partly, this is because the hardware being designed now is so complex that exhaustive simulation is no longer possible, so that other techniques are needed to complement simulation. Another reason for the renewed interest is that, as formal verification techniques have become more sophisticated, they have been shown to be usable on sizable pieces of hardware, not just on toy examples.**

In this paper, we will look at various formal verification techniques and how they can be applied to sequential hardware, especially at the register-transfer level. We will begin with the basic elements of a verification system, as illustrated on the relatively simple problem of verifying combinational circuits. Then we will consider the more complex problems involved in analyzing sequential systems and the techniques that have been developed to solve them. Throughout, we will focus on those techniques whose utility has been demonstrated on real systems, including higher order logic, temporal logic, predicate transformers, state-machine models, and model checkers.

I. Introduction

1.1. Motivation

CORRECTNESS is a major consideration in the design of computers and other digital systems. As we become more and more dependent on computers and digital controllers—in transportation systems, in medical applications, in defense systems, in banking, and so on—the cost of a failure is becoming unacceptably high. It may mean the loss of life or serious impairment for many human beings or the disruption of vital economic and commercial activities.

Designers normally try to ensure correctness through simulation and testing. Essential as they are, these techniques do have their limitations. For a large, complex system, it is impossible to test or simulate all possible inputs or sequences of inputs. Furthermore, simulation and test inputs are usually designed to detect only certain well-defined types of faults. They could miss a subtle design error that might cause unexpected trouble under a particular set of conditions.

Formal verification, on the other hand, attempts to establish universal properties about the design, independent of any particular set of inputs. Typically a formal verification system uses rigorous, formalized reasoning to prove statements of the form:

For all feasible inputs, the behavior of this design is consistent with the behavior required by the specification.

or

For all feasible inputs, the design has property X, which is required by the specification.

Formal verification techniques thus promise to complement simulation and test because the formal techniques can generalize and abstract the behavior of the design, whereas the others cannot. It is analogous to the difference between deriving laws in physics from first principles and doing experiments. One can search for patterns in experiments and surmise that these are indicative of general physical laws, but one can never be sure that one has seen the whole picture. On the other hand, laws derived from first principles can be assumed to hold universally, as long as the initial assumptions are true.

1.2. Organization

In this paper, we will look at the underlying theory and basic techniques being used for the formal verification of digital hardware. Section II is an introduction to the formal verification of hardware. It develops the basic definitions and concepts underlying formal proofs of correctness. It also lists some of the complex problems involved in extending formal verification to real systems. The next three sections describe some of the tools and techniques used in present-day verification systems and gives examples of systems that use them. Section III is on higher order logic, while Section IV considers temporal logic. Section V describes how different techniques for modeling computational structures have been applied to hardware. Finally, Section VI discusses some of the limitations of formal verification and what part it is likely to play in hardware verification.

Throughout the paper, the emphasis will be on basic concepts and methodology and their application to real systems. There is no attempt to give an adequate survey

Manuscript received July 18, 1990; revised June 12, 1992. This paper was recommended by Associate Editor A. Parker.

The author is with the Computer Science Department, Boston College, Chestnut Hill, MA 02167.

IEEE Log Number 9205510.

of all the work that has been done in the field. The systems used as examples have been selected because they are good illustrations of the principles involved and because they have been applied with some success to the verification of relatively large designs.

1.3. Notation

We will frequently use the standard operators for mathematical logic. Let P and Q be any two predicates, that is, expressions that can evaluate to either *true* or *false*. Then $P \wedge Q$, meaning P *and* Q, is true whenever both P and Q are; $P \vee Q$, P *or* Q, is true whenever either P or Q is; and $\sim P$, read *not* P, is true whenever P is false and *vice versa*. Furthermore, $P \supset Q$ is used for P *implies* Q, which is equivalent to $\sim P \vee Q$; and $P \equiv Q$ means P is *logically equivalent* to Q, in other words $P \supset Q$ and $Q \supset P$. Among the logic operators, $\sim$ has the highest precedence, meaning that it binds most tightly; then, in decreasing order, $\wedge$, $\vee$, $\supset$, and $\equiv$.

Some forms of logic also use the two *quantification* operators: $\forall$, *for all*, and $\exists$, *there exists*. $\forall x\ P$ means that P is true for all possible values of x (in some domain implied by the context or explicitly given). $\exists x\ P$, on the other hand, means that P holds for at least one value of x. A quantifier applies to as much of the expression to the right as possible.

More specialized notation will be introduced as needed.

II. Basic Concepts in Hardware Verification

In this section, we will develop the requirements for a formal verification system. First we will look at the basic elements of a verification system for simple combinational logic. Then we will explore the additional issues that must be addressed in the formal verification of sequential systems. Finally, we will list the basic approaches that have been developed to address these problems. Each of these approaches will be described in detail in a later section.

2.1. An Example

The following example shows how formal verification techniques can be applied to a simple combinational circuit. Although the example is presented rather informally, it could easily be made more rigorous.

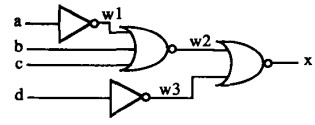


Fig. 1. Circuit to be verified.

Suppoe that, in a digital system we are building, we require a combinational logic circuit with four inputs a, b, c, and d and one output x, where $x = f_S(a, b, c, d)$ and f_S is defined as follows:

$$x = c \wedge d \vee a \wedge b \wedge d \vee \sim a \wedge \sim b \wedge d \vee \sim a \wedge \sim c \wedge d.$$

We are given the circuit in Fig. 1, which, it is claimed, implements the specified behavior. We can verify that the circuit does in fact have the required behavior by writing a logic expression for the circuit's behavior, showing how x depends on a, b, c, and d, and proving that that expression is equivalent to the specified behavior.

One way to extract an expression for the behavior of the actual circuit is to label the internal nodes as shown in Fig. 1 and to write an expression for the output of each of the gates as a function of its inputs:

$$\begin{aligned} w_1 &= \sim a \\ w_2 &= \sim(w_1 \vee b \vee c) \\ w_3 &= \sim d \\ x &= \sim(w_2 \vee w_3). \end{aligned}$$

We assume that the logical behavior of each of the gates is given as a primitive.

Substituting the expressions for the w's into the expression for x in order to eliminate the w's gives the behavior of the proposed implementation: $x = f_I(a, b, c, d)$, where

$$f_I(a, b, c, d) = \sim(\sim(\sim a \vee b \vee c) \vee \sim d).$$

Showing that the implementation is equivalent to the specification thus becomes a matter of proving $f_S \equiv f_I$, which we can do using equivalence rules that are familiar from Boolean algebra:

$$\begin{aligned} f_I &= \sim(\sim(\sim a \vee b \vee c) \vee \sim d) \\ &\equiv ((\sim a \vee b) \vee c) \wedge d && \text{By DeMorgan's Law; } \sim(\sim A \vee \sim B) \equiv A \wedge B \\ &\equiv ((\sim a \vee (a \wedge b)) \vee c) \wedge d && x \vee y \equiv x \vee \sim x \wedge y \text{ with } x = \sim a \text{ and } y = b \\ &\equiv (\sim a \vee (\sim a \wedge \sim b) \vee a \wedge b \vee c) \wedge d && x \equiv x \vee x \wedge y \\ &\equiv ((c \vee \sim a) \vee \sim a \wedge \sim b \vee a \wedge b) \wedge d && \text{Commutativity of } \vee \\ &\equiv ((c \vee \sim a \wedge \sim c) \vee \sim a \wedge \sim b \vee a \wedge b) \wedge d && x \vee y \equiv x \vee \sim x \wedge y \text{ and Commutativity of } \wedge \\ &\equiv (c \vee a \wedge b \vee \sim a \wedge \sim b \vee \sim a \wedge \sim c) \wedge d && \text{Commutativity of } \vee \\ &\equiv (c \wedge d \vee a \wedge b \wedge d \vee \sim a \wedge \sim b \wedge d \vee \sim a \wedge \sim c \wedge d) && \text{Distributing } \wedge \text{ over } \vee \\ &\equiv f_S \end{aligned}$$

This shows that the design and the specification are logically equivalent, meaning that for any set of inputs, the two give the same output.

2.2. *The Structure of a Verification System*

The above example illustrates, in a simplified form, the basic steps involved in formal verification. We began with a specification of the behavior required of the circuit and a description of the structure of the circuit itself. The structural description had to be translated into a formal logic so that we could apply formal proof techniques to it. This could easily be done because at the level at which we are modeling the circuit, wires can only take on two values, which can be identified with the logical values *true* and *false*, and the *and*, *or*, and *not* gates perform the same functions as the corresponding logical operators. The behavioral specification was already written as a logical formula, so no translation was necessary there. Often, however, the behavior is given in another language, such as a high-level programming language or hardware description language, and this must also be translated into the logic. Finally, we used proof techniques for the logic to show that the function implemented by the circuit was the same as that required by the specification. For a purely combinational circuit, this was sufficient to establish that the design met the specification.

This process is summarized in Fig. 2. The process begins with a behavioral specification and a description of the design, which is usually a structural description of some sort, augmented with information about how the individual components work. In general, both descriptions must be translated into formulas in some formal logic. These formulas are then compared using the rules for reasoning in that logic to check whether the implemented behavior corresponds to the specified behavior. In some cases, exact equivalence may be required. In others, the specification may be partial, requiring only that certain properties hold in the design. In such cases it is sufficient to prove that the behavior of the design is consistent with the properties required in the specification. Unfortunately, because the underlying logic may not be decidable, it is not always possible to guarantee that the correspondence checker will give either a yes or no answer in every case. That is why the third output, "Don't know," is shown for the correspondence checker in Fig. 2.

There have been a number of systems that have used this basic approach to verify combinational logic. They differ principally in the method they use for checking logical equivalence or consistency. A number of systems have used rewrite rules, similar to our example, to prove equivalence between Boolean expressions, either by reducing one expression to the other or by showing that the negation of their equivalence reduces to a contradiction. Examples are the early work by Wagner on hardware verification [74] and subsequent systems created by Hanes [32] and Chandrasekhar [10], among others. Roth [69] used a variation of the D-algorithm for Boolean comparison. His algorithm started with corresponding outputs of the two expressions to be compared and worked back to the inputs, trying to find substitutions that would give the outputs different values. If no such substitutions could be found, the expressions were proved equivalent. Other systems, such as the one by Wojcik [75], have used the resolution method [68]. This is an algorithm, often used in automated theorem provers, that seeks to prove a formula by first negating it and transforming it to a canonical product-of-sums form, then reducing it by eliminating common literals until one of the clauses in the product reduces to false. A detailed explanation of the algorithm can be found in Manna [46] and Nilsson [64]. More recently, several verifiers for combinational logic have been based on Binary Decision Diagrams or BDD's [25], [45]. BDD's provide a compact representation for Boolean expressions. Furthermore, Bryant has shown that the BDD for a Boolean function can be reduced to a canonical form, which means that any two equivalent expressions will have the same reduced form [5]. Therefore, comparison of Boolean expressions represented as BDD's is straightforward. The major disadvantage of BDD's is that, like any canonical form, the size of the representation can be exponential in the number of variables.

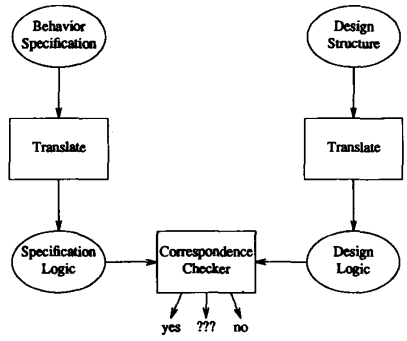


Fig. 2. The verification process.

Yoeli, in a recent book, has given an excellent tutorial on formal verification techniques, especially those applicable to combinational circuits [76].

Another way in which formal verification can be used is to verify the design process itself. If the design is not produced by hand, but is produced directly from the specification by a series of well-defined steps, and if we can show that each step preserves the behavior given in the specification, then we have a proof, within the limits of the model, that the final design has the same behavior as the specification.

Given our specification for the logic example, for instance, the circuit in Fig. 1 might be produced by translating the specification directly into gates and then applying various optimizing transformations to minimize the number of gates. If we could prove that the original translation process always preserves the functionality of the logic specification and that each transformation leaves the

behavior unaffected, that would be sufficient to establish that any design produced by the system had the same behavior as its specification.

This alternative approach is attractive for a number of reasons. First of all, synthesis, especially at the logic level, is coming into general use in industry. In fact, one of the advantages claimed for synthesis is that it is guaranteed to produce correct designs. But this cannot be taken for granted. The synthesis algorithms must be subjected to the same rigorous analysis that we would apply to the design themselves if we were to establish their correctness. Another important advantage is that the verification only has to be done once for the system, rather than for every design produced. Producing a formal proof of correctness is a laborious process under the best of circumstances. If the labor can be amortized over the many designs produced by a synthesis system, it is easier to justify the investment.

2.3. *Further Issues in Formal Verification*

For the example given in Section 2.1, the verification was straightforward. It was a small, purely combinational circuit, where the specification and design were already at the logic level. The simple propositional calculus provided an adequate formal system for proving equivalence, and translating both the specification and the design into that logic was trivial.

Sequential digital systems are far more complex, of course, and require more powerful verification tools. In what follows, we will look at some of the difficulties that must be dealt with in the formal verification of sequential circuits.

2.3.1) *Sequential Behavior*

The typical digital system does more than just map a set of inputs to a set of outputs in a time-independent manner. It has memory, or state, and its behavior at any point depends on the state, which in turn depends on the past history of inputs. Furthermore, most digital systems cannot be described as simply taking some inputs, doing a computation, and giving some outputs. Rather they run continuously, mapping sequences of inputs to sequences of outputs. Formal logic, on the other hand, at least in its simplest forms, does not contain in any fundamental way the notions of time, history, or memory. It simply expresses static relationships.

Consider, for example, a simple serial-to-parallel converter. The specification might be that the circuit waits for the *start* line to be raised, reads 8 bits of input synchronized to an external clock and finally raises the *ready* line and outputs an 8-bit byte, where the leftmost bit of the output is the first bit that was read. There is no straightforward way to translate that specification into simple Boolean equations. Moreover, the implementation requires memory in the form of a shift register or its equivalent to store the bits as they are read in. The concept of reading and holding data so that it is available at a later time is also not easily expressible in the simple logic used in Section 2.1. Even more difficult would be the task of proving the equivalence between the specification and the implementation, since they are built around different concepts. The specification speaks of the dynamic relationship between sequences of inputs and outputs, while the implementation involves a static structure.

2.3.2) *Modeling Time*

Modeling time is a problem that is related to, yet distinct from, modeling sequential behavior. In real gates, the output does not change as an immediate function of the input, as the simple logical model implies. Rather there is a delay from inputs to outputs, and that delay is often significant. This is not just important for doing the timing analysis of a circuit, which could be done separately by using algorithms created specifically for that problem. In some cases, even the logic-level behavior does not make sense unless the delay is taken into account. For example, one would not be able to prove that two cross-connected *nor* gates form an R–S flip-flop without being able to model some delay in the gates. Without that delay, the circuit has no stable solution. Yet simple logic has no straightforward way of expressing time-dependent behavior. There is no obvious way, for example, of expressing the fact that the output y of an inverter is the complement of what the input x was some time Δt previously.

2.3.3) *Levels of Abstraction*

For a complex system, it is often desirable to specify the required behavior at a higher level of abstraction than the logic level. For example, the specification for a computer is usually fixed at the instruction set level. That means that the effect of each instruction on the memory and certain key processor registers is determined; but it is not necessary, or even desirable, to specify what happens to each flip-flop on each microcycle.

The structure also may be described at different levels. The implementation might be given at the register-transfer level, as a set of interconnected registers, memories, ALU's, shifters, and multiplexers, along with a PLA or microcoded controller. Or it might be a gate-level structure, or even a network of transistors, that is to be verified against the specification.

The need to deal with different levels of abstraction creates a number of difficulties. First of all, the verification system must be able to handle a richer set of data types than just Booleans. A high-level specification of behavior involves at least bit vectors, and usually integers. In some cases, signal processing for instance, it might even use floating point numbers. Either these data types must all be translated into Booleans, so that a simple logic can handle them, or the formal system must be powerful enough to reason about the high-level data types directly. The former often leads to unwieldy expressions, while the latter requires a much more elaborate proof system.

The second problem is that the specification and the implementation are likely to be represented at different levels of abstraction. In that case, the correspondence between the two is not as obvious as it would be if both were given at the logic level. The implementation usually contains a great deal more detail than the specification, and there must be some way of abstracting from all that detail so that the two can be compared. For example, the specification for a CPU, as noted earlier, would probably only make reference to certain key registers and to major cycles of the machine. On the other hand, the implementation would be described at a level no higher than the microarchitecture, which would include many internal registers not mentioned in the specification, and would execute several microcycles for each major cycle. Worst yet, in a high-performance design, such as a pipelined machine, there might not be a single register in the microarchitecture that corresponds to what is described as a single register in the specification. For example, there might be several instances of the instruction register alive at once. Furthermore, although the specification might be written as if the major cycles executed sequentially, in the actual design they might overlap. When verifying a design against a higher level specification, either the user must designate the points in time when the descriptions should be compared, or the verification system must somehow discover them. Similarly, the correspondence between registers must also be given.

Finally, at higher levels of abstraction, the translation from static structure to function is not as obvious as it is at the logic level. In the example in Fig. 1, one could simply compose the functions for the individual gates to get the function for the whole circuit. But when there are a number of complex blocks interconnected by shared buses, with all of the parts driven by externally supplied control signals, extracting the behavior is not so straightforward. Intuitively we may be able to describe how to do it, but formalizing the process of translating a netlist into dynamic behavior is not a simple task.

2.4. *Extensions to the Basic Method*

As the above discussion indicates, verifying real designs makes many demands on the underlying formal system. It must be able to handle data types, such as the integers, and complex objects, such as arrays and time-dependent sequences; and it must be capable of describing hierarchy and structure and relating these to behavior. There are three basic ways to extend the formal system to meet these demands. First, we can use a higher order logic that is powerful enough to allow us to define and reason about the complex objects involved in verification. Second, we can extend the logic by adding new operators and rules specially adapted to reasoning about sequences and structure. And third, we can imbed the logic in another system that is designed specifically to handle some of the issues that are difficult to represent in logic. In the next three sections we will look at each of these approaches, presenting both the underlying ideas and examples of how they have been used in practice.

III. Higher Order Logic

There is a hierarchy of logics for formal reasoning, arranged according to the generality of their data types and operators [46]. For the example in Fig. 1, a simple propositional logic was adequate. In propositional or *zeroth-order* logic, only propositional variables are allowed, that is, variables over {*true*, *false*}; and the only operators or functions allowed are the logical operators $\wedge$, $\vee$, $\sim$, and other operators derived from these. Propositional logic can be extended in a small way by adding the *quantification* operators $\forall$ and $\exists$.

First-order logic is much more general than propositional logic in that it allows variables over one or more other types. It also allows constant operators, functions, and predicates over the added types. An example of a first-order formula is

$$(x < y - 1) \supset (y \neq 0)$$

where x and y are nonnegative integers.

First-order logic can be extended by adding axioms defining certain types and the operators over them. One very useful extension is to add rules for equality, including a rule that substitution of equal expressions preserves equivalence. Another is to add axioms defining the natural numbers (zero and the positive integers). One of the axioms for the natural numbers that will prove to be important in many contexts is the so-called "principle of mathematical induction:"

> For any predicate P over the natural numbers, if $P(0)$ is true and if $P(x) \supset P(x + 1)$ for all x, then $P(x)$ holds for all x.

According to the principle of induction, we can prove any proposition P over the natural numbers by proving (1) $P(0)$ is true, and (2) if $P(x)$ is true, it follows that $P(x + 1)$ is also true. This principle generalizes to any countable set with a least element.

Second-order logic is more general than first-order, in that it allows variables and operators over functions and predicates. Therefore, functions and predicates can be defined and manipulated as objects in themselves, not just used to define and manipulate simpler objects. One advantage of second-order logic is that it does not have to be extended with special theories for certain types and operators. The logic is general enough so that these theories can be developed within the logic itself. For example, the statement of the principle of induction in second-order logic is

$$\forall P((P(0) \wedge (\forall x\, P(x) \supset P(x + 1))) \supset (\forall x\, P(x))).$$

Because second-order logic allows function variables, and therefore can reason directly about functional objects, it

is powerful enough in itself to handle the key issues related to hardware verification. As we have seen, we can build up theories of types such as the natural numbers in a second-order logic. Arrays, which can be used to represent bit vectors, for example, can be expressed as functions mapping some index set into the set of possible values. And time sequences can be expressed as functions from a time variable into the set of possible values. Suppose, for example, that the output port *oport* for a particular piece of hardware is eight bits wide and produces a continuous sequence of outputs. Then the action of that port can be modeled as a function $oport(t, i)$, that maps t, an integer representing time, and i, an index from 1 to 8 representing a particular bit of *oport*, into the set $\{T, F\}$ or $\{1, 0\}$, depending on how we wish to represent bits. Thus $oport(t_0, 1)$ represents the first bit of *oport* at time t_0.

When second-order logic is used for hardware verification, a circuit at any level in the hierarchy is represented as a relation between one or more input functions and one or more output functions, where the relation is expressed as a formula in the logic. For example, a simple inverter with a delay of t_d time units could be represented by the predicate:

$$inverter(in, out) =_{\text{def}} \forall t\ out(t + t_d) = \sim in(t).$$

This says that the behavior of the inverter is properly described by an input function $in(t)$ and output function $out(t)$ such that the output at each $t + t_d$ is the complement of what the input was at t. The functions *in* and *out* are listed as parameters to *inverter* because they represent the external connections to the circuit. If t_d were considered variable, it would also be a parameter.

Circuits are put together to form larger circuits by ANDing together the expresions for them. Connections between the component circuits are shown by substituting a common variable name for each of the parameters that are connected together. For instance, if we have circuit $C_1(in_{11}, in_{12}, out_{11})$ and circuit $C_2(in_{21}, in_{22}, out_{21})$, and we want to connect out_{11} of C_1 to in_{22} of C_2, we can write $C_1(in_{11}, in_{12}, x) \wedge C_2(in_{21}, x, out_{21})$. This means that the circuit operation is only properly described by sets of values such that out_{11} of C_1 and in_{22} of C_2 have the same value. We can hide an internal connection such as x by existentially quantifying it. This requires that some common value exist, but that value is not part of the input or output. For example, suppose we have definitions of a delayless *nor* gate and a delayless inverter:

$$nor(a, b, c) =_{\text{def}} c = \sim(a \vee b)$$

$$inv(x, y) =_{\text{def}} y = \sim x.$$

We can form an "or" circuit by connecting the output of a *nor* gate to an inverter:

$$or(a, b, c) =_{\text{def}} \exists x\ nor(a, b, x) \wedge inv(x, c).$$

The following proof shows that this does indeed have the behavior of an *or* gate:

$$\begin{aligned}
&\exists x\ nor(a, b, x) \wedge inv(x, c)\\
&\quad \equiv \exists x\ (x = \sim(a \vee b) \wedge c = \sim x)\\
&\quad \equiv \exists x\ (x = \sim(a \vee b) \wedge \sim c = x)\\
&\quad \equiv \exists x\ (\sim c = \sim(a \vee b) \wedge \sim c = x)\\
&\quad \equiv \exists x\ (c = (a \vee b) \wedge \sim c = x)\\
&\quad \equiv c = (a \vee b) \wedge \exists x\ \sim c = x)\\
&\quad \equiv c = (a \vee b)
\end{aligned}$$

The next-to-the-last step is valid because a quantifier can be moved past an expression that does not contain the quantified variable. The last step can be taken because $\exists x\ x = t$ for some term t is always true. In general this means that $\exists x\ (P(x)(x = t))$ is equivalent to $P(t)$, which is often useful for simplifying the expressions for circuits created by composing other circuits and hiding the connections.

To illustrate how a higher order logic is used, we analyze a simple version of the series-to-parallel converter example mentioned earlier, proving that the implementation behaves as specified. Our version of the converter has a primitive asynchronous interface. When the *start* line is raised, the converter reads 8 bits, then raises the *ready* line and outputs the 8 bits on parallel output lines. We assume that the inputs are synchronized to an external clock, which is represented by a time variable t, ranging over the positive integers. Each unit of t represents one clock tick.

To simplify the specification and the proof, we introduce some additional notation. The conditional operator $(b \rightarrow t_1, t_2)$ selects between two terms t_1 and t_2 depending on the value of the boolean b. It can be read as "if b then t_1 else t_2." We also define the predicate *range* by

$$range(i, m, n) =_{\text{def}} (i \geq m \wedge i \leq n).$$

Thus for example $range(i, 1, 10)$ means that i is between 1 and 10, inclusive. Finally we represent individual bits as boolean variables, that is, they are variables over $\{T, F\}$ so that boolean operators can be applied to them directly.

With these assumptions, we can specify the required behavior of the series-parallel converter:

$$\begin{aligned}
(S1)\quad &spconvert(start, in, out, ready, t)\\
&=_{\text{def}} start(t) \supset (\forall i\ (range(i, 1, 8)\\
&\supset (\sim start(t + i) \supset (\sim ready(t + i)\\
&\wedge ready(t + 9) \wedge out(i, t + 9)\\
&\quad = in(t + 9 - i))))).
\end{aligned}$$

This says that if *start* is asserted at time t and not for the next eight clocks, then *ready* will be asserted at the $(t + 9)$th clock and not before, and the 8 bits of the output at

$t + 9$ are the bits read at the input from $t + 1$ through $t + 8$.

The converter is implemented with a counter and a shift register, as shown in Fig. 3. The counter loads a value *cin* when the start line is asserted and counts down one for each t until it reaches 0, at which time it asserts *ready*. The shift register always shifts a bit in from its input, so that the output is always the last eight input bits. These are defined by the following expressions:

(D1) $counter(start, cin, count, ready, t)$

$$=_{def} (count(t + 1) = (start(t) \rightarrow cin(t), count(t) - 1)) \wedge (ready(t) = (count(t) = 0 \rightarrow T, F))$$

(D2) $shifreg(srin, srout, t)$

$$=_{def} srout(1, t + 1) = srin(t) \wedge \forall i\ range(i, 2, 8) \supset srout(i, t + 1) = srout(i - 1, t).$$

Proving the correctness of the implementation means proving that the implemented behavior implies the specified behavior:

$$counter(start, 8, count, ready, t) \wedge shifreg(in, out, t) \supset spconvert(start, in, out, ready, t).$$

Note that by substituting *in* for *srin* in the shift-register predicate, we implicitly identify *in* with the shift-register input. Other inputs and outputs are handled similarly. We also input a constant 8 to the counter.

What we want to prove is that the three basic characteristics of the specified behavior, $\sim ready(t + i)$ for i from 1 to 8, $ready(t + 9)$, and $out(i, t + 9) = in(t + 9 - i)$ follow from the definition of the implementation and certain initial assumptions about the inputs and internal variables. To carry out the proof, we use the fact that we can prove $A \supset B$ by assuming A and deriving B from it. We also use the fact that, if $A \supset B$ and $A \supset C$, then $A \supset B \wedge C$. We will proceed, therefore, by assuming that all the predicates defining the implementation in D1 and D2 and all the initial assumptions for the specification S1 hold. This gives us the following assumptions:

(A1) $count(t + 1) = (start(t) \rightarrow cin(t), count(t) - 1)$, from D1

(A2) $ready(t) = (count(t) = 0 \rightarrow T, F)$, from D1

(A3) $srout(1, t + 1) = srin(t)$, from D2

(A4) $\forall i\ range(i, 2, 8) \supset srout(i, t + 1) = srout(i - 1, t)$, from D2

(A5) $start(t)$, assumed from the initial conditions for S1

(A6) $range(i, 1, 8)$, assumed from the initial conditions for S1

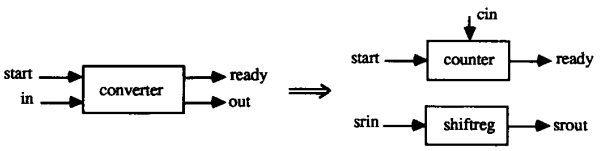


Fig. 3. Series-parallel converter.

(A7) $\sim start(t + i)$, assumed from the initial conditions for S1

From these we derive the conditions on *ready* and on *out* in S1.

We prove the conditions on *ready* by proving the following more general property:

(I1) $counter(start, N, count, ready, t) \supset (start(t) \supset (\forall k\ (range(k, 1, N) \supset \sim start(t + k)) \supset count(t + k) = N - k + 1))$.

This is done inductively. That is, we first show that it is true for the base case $k = 1$. Then, assuming it is true for k, we show that it is true for $k + 1$.

1) **Base Case ($k = 1$):** From A1, $start(t)$ implies that $count(t + 1) = N$, which of course is just $N - k + 1$.
2) **Inductive Step:** Assume that $count(t + k) = N - k + 1$. From the assumptions in (I1), if $range(k + 1, 1, N)$ then $\sim start(t + k + 1)$ and from A1, $count(t + (k + 1)) = count(t + k) - 1 = N - k + 1 - 1 = N - (k + 1) + 1$, which proves I1 for $k + 1$.

From property (I1) with $N = 8$, we have for $i < 9$, $count(t + i) \neq 0$ and $count(t + 9) = 0$. From A2, it follows that $\sim ready(t + k)$ holds for k less than 9 and $ready(t + 9)$ is also true.

The condition on *out*, $out(i, t + 9) = in(t + 9 - i)$, is proved in much the same way. We first prove the property:

(I2) $shiftreg(srin, srout, t) \supset srout(i, t) = in(t - i)$.

This is also done inductively.

(1) **Base Case ($i = 1$):** From A3, $srout(1, t) = in(t - 1)$.
(2) **Inductive Step:** Assume that $srout(i, t) = in(t - i)$. Then $srout(i + 1, t) = srout(i, t - 1)$ from A4, which is equal to $srin(t - 1 - i)$ by the inductive assumption, substituting $t - 1$ for t. This, of course, is equal to $srin(t - (i + 1))$.

This establishes I2. From I2 with *out* = *srout* and *in* = *srin*, we can substitute $t + 9$ for t to get $out(i, t + 9) = in(t + 9 - i)$, which completes the proof.

To show how iterative structures can be described in this logic, we extend the example by defining the structure

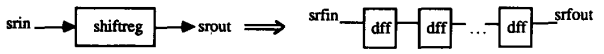


Fig. 4. Implementation of a shift register.

of a shift register as an array of D flip-flops, as shown in Fig. 4, and proving that that implementation has the behavior required of the shift register. To remove unnecessary complications in the example, we assume no control lines for the flip-flops except the implicit clock, so that each flip-flop loads its input on every cycle. This gives the definition:

(D3) $dff(fin, fout, t) =_{\text{def}} fout(t + 1) = fin(t)$.

A shift register is constructed from a series of such flip-flops, with the input to flop-flop 1 connected to the shift register input and the input to flip-flop k tied to the output of flip-flop $k - 1$ for $k > 1$. The kth output of the shift register, of course, is the output of the kth flip-flop.

Just as a sequence of actions over time can be defined recursively, so can a sequence of similar structures. Thus the most straightforward definition of the shift-register structure has a recursive form. We define a 1-bit shift register as a single flip-flop and an $n + 1$-bit shift register as an n-bit shift register with a flip-flop appended to it. The predicate for the n-bit shift-register structure, called $srf(n, srfin, srfout, t)$ is then defined by the following:

(D4) $srf(1, srin, srout, t)$

$$=_{\text{def}} dff(srin, srout(1), t)$$

$$srf(n + 1, srin, srout, t)$$

$$=_{\text{def}} srf(n, srin, srout, t)$$

$$\wedge\ dff(srout(n), srout(n + 1), t).$$

Here we use $srout(i)$ to mean the ith "bit" of $srout$, where in fact each "bit" of $srout$ is a time sequence. In other words, $srout(i)(t) = srout(i, t)$. Note that the flip-flops are connected together by making the input to the $(n + 1)$th flip-flop the same variable as the output of the nth, $srout(n)$.

Proving the correctness of the implementation involves proving that the high-level description of the shift register, given in D2 above, follows from the definition of the lower level implementation given in D4. In other words, we must prove that

$$srf(8, srin, srout, t) \supset shiftreg(srin, srout, t).$$

One simple way to do this is to prove the following somewhat more general inductive property:

(I3) $srf(n, srin, srout, t) \supset srout(1, t + 1)$

$$= srin(t) \quad \text{(I3a)}$$

$$\wedge\ \forall i\ range(i, 2, n) \supset srout(i, t + 1)$$

$$= srout(i - 1, t) \quad \text{(I3b)}.$$

Not only is this property easier to prove, but it is more general, because it applies to any size shift register. Property $I3$ is proved by induction on n.

(1) **Base Case ($n = 1$):** $srf(1, srin, srout, t) = dff(srin, srout(1), t) = srout(1, t + 1) = srin(t)$ by definition D4, which establishes I3a. I3b is trivially true, since the range 2 to n is empty.

(2) **Inductive Step:** Assume that I3 holds for n. $srf(n + 1, srin, srout, t)$ implies $srf(n, srin, srout, t)$ by definition D4, from which I3a and I3b for i from 2 to n both follow by the inductive assumption. We only need to prove $srout(i, t + 1) = srout(i - 1, t)$ for $i = n + 1$. Now $srf(n + 1, srin, srout, t)$ implies $dff(srout(n), srout(n + 1), t)$ by D4. By D3 this is $srout(n + 1, t + 1) = srout(n, t)$, which is I3b with $i = n + 1$.

The specification $shiftreg(srin, srout, t)$ follows easily from I3 with $n = 8$.

This example illustrates how second-order logic can handle many of the complex issues involved in hardware verification. It can describe and verify sequential behavior and relate it to structure, and it can work across different levels of the structural hierarchy. Of course, the example is much simplified. It did not show the fine details of timing, for one thing. The implementation of the shift register as a series of flip-flops chained together has possible race problems. These do not show up in the proof because the model of time, with a clock cycle chosen as the smallest unit, is too coarse. This is a fault of the way we modeled the circuit, not a fundamental limitation of second-order logic. We could have made the time unit much smaller and added assumptions about the earliest and latest time the clock could reach each flip-flop, about maximum and minimum delays through the flip-flops, and so on. All of this is expressible in the logic, although the descriptions of the hardware and the proofs would be far more complex.

A fine-grained model of time would also allow the description of signal transitions as well as signal levels. This was not possible in the unit-cycle model used in the example. Since the signals are, in effect, sampled just once per clock cycle, it is only possible to describe their levels during each cycle. Using a smaller unit of time, however, it is possible to assert that a signal changes at a certain time. We could, for example, formulate a predicate $posedge(x, t_0)$, which means that a signal x goes from F to T at time t_0:

$$posedge(x, t_0) =_{\text{def}} \sim x(t_0) \wedge x(t_0 + 1).$$

We could then use this to give a more precise definition of a positive edge-triggered D flip-flop with a fixed delay t_d:

$$DFF(din, dout, clk)$$

$$=_{\text{def}} \forall t\ dout(t + t_d)$$

$$= (posedge(clk, t) \rightarrow din(t), dout(t + t_d - 1)).$$

See Melham [51] for a good explanation of how timing is described in higher order logic.

Another limitation of the example is that, while it did show the use of natural numbers in the definition of the counter, it did not show how a specification formulated in terms of natural numbers can be related to an implementation formulated in terms of bits. If we had gone on to describe and verify a logic-level implementation of the counter, we would have seen how this could be done.

One thing the example illustrates quite well is the prevalence of repetitive structures in both the temporal and spatial descriptions of hardware. Behavior is best described as sequences of actions in time, and many hardware structures consist primarily of arrays of similar elements tied together in very regular ways. The example also shows the importance of recursive definitions for specifying these regular structures and of induction for verifying them. In fact, induction has turned out to be the primary proof technique in this and many other approaches to verification, precisely because it is so well adapted to proving properties of sequences. Often the most important step in a proof is finding the right inductive property, also called an *invariant* because it holds for all elements of the sequence. Once the right invariant is found, the proof becomes much simpler; but finding it often requires considerable insight into why the property being verified is true. It is the one step in the proof process that, more than any other, has defied mechanization.

The proofs in our example seem quite simple, and in the end they are. Yet even these proofs embody several important insights about the hardware being verified. For example, the proof that $ready(t + 9)$ is asserted does not work unless we can assume $\sim start(t + i)$ for i from 1 to 8. This shows that in using the circuit as designed, we would have to ensure that *start* was asserted once at the beginning of the conversion cycle and locked out the rest of the time. If this could not be guaranteed, we would have to redesign the circuit to insulate it from the start line after the initial signal was detected. The verification also helps to establish the correct boundary conditions for the shift-and-count loop. It shows that, given the timing model used, eight must be loaded into the counter initially rather than seven or nine, and the parallel output is available on the ninth cycle rather than the eighth. The reason the proofs seem so simple is that we got the specification and design right. Often the greatest value of verification is in showing where the implementation and specification are not consistent and suggesting how they need to be corrected.

There are several working systems that use second-order logic or its equivalent for hardware verification. The Higher Order Logic (HOL) system of Gordon et al. [7], [29] uses second-order logic to model and verify hardware in a way that is similar to what we have shown in our examples. In fact, the examples and explanations given above owe much to Gordon's system. HOL is based on type theory. That is, every variable must be defined to have a certain type, and it can only take on values of that type. Thus a boolean variable cannot take on the values 1 or 17, nor can an integer be *true*. The typing extends to higher order objects, such as functions. If a function is defined as mapping an integer to a Boolean, it cannot take another function as an argument. This minimizes ambiguity, errors, and confusion in specifications and proofs, much like typing in a programming language. It also avoids the worst decidability problems, because it does not allow a predicate to take itself as an argument.

HOL has a very general theorem prover that allows the user to add new types and build up theories about them by proving theorems and adding them to the rules available for use in further proofs. The theorem prover supports induction also. One problem is that it requires a great deal of guidance from the user. The user must formulate the theorems to be proved and provide most of the steps along the way [71]. Thus using the system is very time-consuming and requires a great deal of expertise, as the developers themselves have noted. Nevertheless, the HOL system has been used to verify several substantial circuits, including two microprocessors [2], [12], [13]. Normally, the specifications and implementations are written directly in the logic, but it is also possible to write them in a hardware description language and translate them automatically into the logic.

Other examples of verification systems based on second-order logic are LCF_LSM [36], [30], the precursor of HOL, and the VERITAS system [33].

A somewhat different approach to the use of logic in verification has been taken by Hunt [39]. He used the Boyer–Moore theorem prover to verify the FM8501, a microprocessor with a large instruction set. Hunt proved that a register–transfer level design of the FM8501, described in terms of what happens during each microcycle, correctly implemented an instruction-set level specification of the microprocessor. He did this by showing that after each macroinstruction execution, which corresponded to several microinstructions, both descriptions had the same effect on that part of the machine state that was visible to the user, such as the register file and the flags.

The Boyer–Moore logic is the first-order predicate calculus with equality. The theorem prover provides some of the power of second-order logic, in that the user can define new data types and add axioms about them. It also has built into it knowledge about how to do general induction. Nevertheless, the fact that functions cannot be manipulated and passed as arguments limits the system somewhat. For example, the system cannot deal directly with time sequences; they must be simulated. What Hunt does is to build lists of values for inputs and outputs, then pull values off the input lists when needed and add output values to the output lists as they are generated. Moreover, instead of proving that the specification and implementation represent the same mapping of inputs to outputs for all time, Hunt assumes that one macroinstruction cycle corresponds to a certain number N of microcycles, and proves that for any inputs and any state of memory, one

macroinstruction cycle has the same effect as N microcycles. There is an implicit inductive argument that if both descriptions have the same effect after one macroinstruction cycle, they will have the same effect after any number of them.

The advantage of the Boyer–Moore theorem prover is its efficiency and ease of use. The user must break down the problem into a series of small theorems that lead to the desired result, but once a theorem is posed, the theorem prover proceeds on its own, using a powerful set of heuristics to guide the proof. Thus while the user must still know a great deal about theorem proving and about the system being verified in order to choose what theorems to prove, the Boyer–Moore system requires less guidance than HOL, at least for those problems that can be formulated inductively. According to one study, the Boyer–Moore system required considerably less user time than HOL to generate a comparable proof [71].

German has also used the Boyer–Moore theorem prover to verify hardware [26].

IV. Temporal Logic

Another approach to reasoning about sequential behavior is to extend the logic so that it can directly describe temporal behavior. This removes the need to make everything an explicit function of time, in some cases leading to simpler specifications of sequential behavior. It also makes it unnecessary to use second-order logic, which can result in a simpler, more efficient proof system.

One way of extending the logic that is often used in hardware verification is *temporal logic* [41], [67]. Temporal logic is derived from modal logic, which is a way of reasoning about a situation where there are many possible states, or "worlds," and predicates can be true in some states and not in others. The two modal operators are $\Box$, read "always" and ∇, read "sometimes." For a predicate P, $\Box P$ means that P is true in every possible state, and ∇P means that P is true in at least one state. It is possible to give this a temporal interpretation that is useful for proving properties about programs or hardware. Given a designated present state, other states are possible future states. In this view, $\Box P$ means that P will always be true in the future, and ∇P means that P will eventually become true. If the states are linearly ordered, then each state has a unique next state, which allows the $\bigcirc$ or "next" operator to be defined. $\bigcirc P$ means that P will be true in the next state to the present one. Other operators can also be defined. For example, P Until Q means that Q will eventually become true, and P holds in all states up to the one in which Q becomes true.

To see how these operators work, consider the following example:

$$s := \mathit{false},\ c := i1,\ b := \sim i2;$$
$$s := \mathit{true};$$
$$a := b \wedge c.$$

The three lines are executed in sequence, while the three statements on the first line are done in parallel. Each assignment represents a Boolean transfer. The variables s, a, b, and c represent flip-flops, while $i1$ and $i2$ represent input lines. These are three temporal states to be considered, one at the end of each statement, if we ignore the initial state in which everything is undefined. Each state is characterized by the values of the variables at that point in the execution. a is set to $b \wedge c$ at the end, so we can validly state that $\nabla (a = b \wedge c)$. On the other hand, $\Box (a = b \wedge c)$ is not valid, since there are some states for which that is not necessarily true. We can be a little more precise about when a is set with the statement $(\sim s \wedge \bigcirc s) \supset \bigcirc\bigcirc (a = b \wedge c)$. This states that if s is false in the current state but true in the next, then a will equal $b \wedge c$ in the state after the next. Note that the condition $(\sim s \wedge \bigcirc s)$ precisely defines one possible present state, so that the consequence $\bigcirc\bigcirc (a = b \wedge c)$ need only be true in terms of that one state. The overall behavior of the system can be expressed as follows:

$$(\sim s \wedge (in1 = x) \wedge (in2 = y)) \supset \nabla (s \wedge (a = x \wedge \sim y)).$$

It is necessary to use the variables x and y in order to capture the values of $in1$ and $in2$ in the initial state. It would not be valid to assert that $a = in1 \wedge in2$, since $in1$ and $in2$ might have changed their values by the time the final state is reached.

The temporal operators and the axioms defining them can be added to the first-order predicate calculus to give first-order temporal logic. Rules of inference are also added for the construction of temporal logic proofs. One such rule is a general principle of induction over states:

$$\text{If } A \supset B \text{ and } A \supset \bigcirc A \text{ then } A \supset \Box B.$$

This says that if A implies B and A is invariant, in the sense that if A holds in one state it also holds in the next, it follows that if A holds in the present state, B holds in that state and all subsequent states. Thus a strategy for proving that some property B always holds is to find an inductive invariant A that implies B. We prove that A holds in some initial state. Then we assume that A holds in a state S and, based on that, prove that A also holds in the next state. This proves that A is true everywhere from the initial state on, and, therefore, so is B. This is similar to the proof by induction on the time variable that we did in higher order logic.

The temporal operators make it easy to express behavior across time. For example, we could define a *nor* gate with a unit delay by the expression:

$$nor(in1, in2, out) =_{\text{def}} (in1 = a) \wedge (in2 = b) \supset \bigcirc (out = \sim (a \vee b)).$$

If the delay was n "states," we could replace the one $\bigcirc$ by n $\bigcirc$'s in a row, abbreviated by $\bigcirc^n$. If we did not want

to specify the delay precisely, we could write

$$nor(in1, in2, out) =_{\text{def}} (in1 = a) \wedge (in2 = b) \supset \nabla(out = \sim(a \vee b)).$$

The behavior of a D flip-flop with a single data input and a level-sensitive clock input could be defined by the following:

$$dff(Din, Clk, out) =_{\text{def}} (((Din = x) \wedge Clk) \supset \nabla(out = x)) \wedge (((out = y) \wedge \sim Clk) \supset \bigcirc(out = y))).$$

Here the delay through the flip-flop is left indefinite. As with the *nor* example, we could specify it more precisely by substituting $\bigcirc^n$ for the ∇.

By defining the time between states to be a small enough unit, we can talk about much finer temporal behavior. For instance, we could define the "rising edge" operator $\uparrow$ by

$$\uparrow X =_{\text{def}} \sim X \wedge \bigcirc X.$$

(This puts the rising edge between the present state and the next.) Using this definition, we could define an edge-triggered flip-flop by substituting $\uparrow Clk$ for Clk in the definition for *dff*.

Temporal logic is especially well adapted to stating and proving so-called *liveness* properties. These properties have to do with the reachability of certain states in a hardware description (or program). For example, one might want to prove that if a certain predicate P is true, the machine will eventually reach a state where Q becomes true. The statement of that in temporal logic is quite simple:

$$P \supset \nabla Q$$

or it might be required that a certain state characterized by P must occur infinitely often. The specification for this is:

$$\Box\nabla P.$$

Because first-order temporal logic does not have functional variables, it is more difficult to formulate statements relating sequences of outputs to sequences of inputs, especially if there is a complex temporal relation between them. The example of the serial-to-parallel converter discussed previously is a good illustration of that. It is simple enough to state the liveness property that once the *start* signal is given and then held off, the *ready* signal will eventually be asserted:

$$(start \wedge \bigcirc(\sim start \text{ until } ready)) \supset \nabla ready.$$

The requirement that when *ready* is asserted, *out* contains the bits read at *in* over the previous eight cycles is not as straightforward to express. One way to do it is to define *out* as a bit vector and to create an index i that can be used to label each input bit as it is read. This leads to one possible expression for the behavior:

$$converter(start, in, out, ready) =_{\text{def}} \exists i ((start \supset (i = 0)) \wedge ((i = k \supset \bigcirc(i = k + 1)) \wedge ((range(i, 1, 8) \wedge \sim start \wedge (y = in)) \supset \nabla(ready \wedge (out[9 - i] = y)))).$$

Here i is not part of the hardware, but is rather a construct to facilitate the description. As we did with second-order logic, we could go on and define the individual components of the implementation and prove that they implied the same behavior as the behavioral specification. This would be done as before, using inductive invariants.

Temporal logic is most often used for proving properties about asynchronous circuits, that is, circuits in which some or all of the signals are not tied to a regular clock. The ability to express properties about the occurrence and temporal ordering of events without exactly expressing their dependence on time is especially useful in such cases. Bochmann, for example, has used temporal logic to specify the behavior of an arbiter circuit and to prove that an implementation of that circuit is correct with respect to the specification [3].

A variation of temporal logic, called *Interval Temporal Logic* (ITL), has been used by Moszkowski for hardware verification [62], [31]. In ITL, predicates are evaluated over intervals rather than in single states. An interval σ is a series of instants $\langle t_m, \cdots, t_n \rangle$. A predicate P is true over an interval σ if P is true for each of the instants in the interval. From this basic definition, Moszkowski has built up a rich set of operators, in addition to the usual $\Box$, ∇, and $\bigcirc$. These operators can be used to describe in detail the temporal behavior of hardware. For example, a clocked D flip-flop with a setup time s and propagate time p can be described by

$$DFF(Din, Clk, Dout) =_{\text{def}} (\uparrow^{s,p} Clk \wedge (Clk \; blk \; Din)) \supset Din \rightarrow Dout.$$

$\uparrow^{s,p} Clk$ states that there is a rising edge of the clock during the interval, with the clock initially low for at least s units before the transition and then remaining high for at least p units after it. *Clk blk Din* means that *Din* must remain stable during the interval until *Clk* changes. Together these conditions require that *Din* must be stable for s, the setup time, while *Clk* stays low; then *Clk* must rise and stay high for at least the propagation time. If these conditions are met, then the value at *Din* is transferred to the output *Dout*.

More complex hardware components and abstract behaviors can also be expressed in ITL, and the logic can be used to verify the designs of circuits against their specifications. Moszkowski, for example, has verified a clocked multiplier and an AM2901 bit-slice chip using this formalism [61]. More recently, Leeser has used ITL to prove the correctness of a number of CMOS circuits

[43]. The proofs depended on the proper modeling and analysis of complex timing behavior.

Other verification systems have used temporal logic for specifying the behavior of a system, but have used different methods for verifying the design against that behavior. Some of these will be considered in the next section.

V. Computational Structures

The last type of verification method we will consider combines logic with another analysis method. Logic is used to analyze and compare the static states of the system, a task for which it is well-adapted, but a different technique is used to trace the dynamic flow from state to state. We will consider four different approaches here. The first uses techniques borrowed from program verification; the second is based on state machine analysis; the third is a hybrid approach that combines state-machine analysis with temporal logic. The fourth approach uses an algebra of recursive expressions to represent the behavior of communicating processes.

5.1. Predicate Transformers

This approach is very similar to program verification techniques [24], [37], from which it was originally adapted. It therefore assumes that the hardware is represented in a form similar to that of a program in a procedural programming language such as Pascal. This means, specifically, that the behavior of the hardware is represented as a set of statements. The primitive statements are assignments to variables and perhaps procedure calls and "goto" statements. These are put together into larger, composite statements using the standard structures of sequential and, in some cases, parallel composition, conditionals, and loops. The basic idea is that a predicate called the *precondition* is attached to the beginning of each statement, and another predicate called the *postcondition* is attached to the end. The precondition describes the state of the system before the statement is executed, and the postcondition describes the state after. For each type of statement in the language, there are rules about how the precondition is related to the postcondition. Thus each statement can be viewed as a *predicate transformer* that maps a precondition into a postcondition or, *vice versa*, a postcondition to a precondition.

There are two basic approaches that can be taken to analyzing programs or hardware descriptions within this framework. One starts with the known state at the beginning of the description and pushes predicates forward from preconditions to postconditions, finally producing a description of the final state or the state at key points. This can then be compared with the specification to see if the behavior satisfies the specification. The other approach begins with the desired state at the end of the description or at key points. This is then pushed back from postconditions to preconditions, giving the initial conditions that are necessary to produce the required behavior. If these initial conditions are consistent with the actual initial state of the hardware, then it can be inferred that the hardware meets the specification.

For the purposes of explanation, we will focus here on the first method, forward propagation. This is the method that seems to be used most often in hardware verification. The opposite approach is similar, though more elegant in its formulation and at the same time less intuitive. One form of the latter approach that is often used in program verification is the axiomatic program logic of Hoare [11], [37], [38].

We first need to give the rules relating preconditions and postconditions for each of the statements in the language. For the sake of simplicity, we assume that the hardware is described in a Pascal-like language that has assignments, noops, conditionals, sequential composition, and **while** loops. The method can easily be extended to include other constructs commonly found in procedural languages.

The assignment statement is the statement that actually changes the internal state of the system by changing the values of some of the variables, and the postcondition must reflect this. For an assignment statement "$x := e$" with x a variable and e an expression, and a precondition P, the postcondition Q is

$$\text{(AS)} \quad Q = \exists x' \, P\langle x \leftarrow x' \rangle \wedge x = e\langle x \leftarrow x' \rangle$$

where $F\langle x \leftarrow y \rangle$ means F with y substituted for every occurrence of x. Essentially we create a new variable x' to stand for the old value of x, carry forward the precondition, with x' substituted for x and append the new fact that $x = e$, again with x' substituted for x in e. If x is not present in P or e, then this reduces to $P \wedge (x = e)$. For example, if the precondition is $y = 2$ and the assignment statement is "$x := 1$," the postcondition is $y = 2 \wedge x = 1$. Using the same precondition and the assigment "$x := y - 1$," the postcondition would be $y = 2 \wedge x = y - 1$, which of course reduces to $y = 2 \wedge x = 1$.

If P and/or e is dependent on x, the situation is more complicated. However, if we can find a function that computes the inverse of e, the expression can still be simplified. To see this, we write P and e with an explicit dependence on x, i.e., $P(x)$ and $e(x)$. Let g be the function such that if $y = e(x)$ then $x = g(y)$. Then $\exists x' \, P(x') \wedge x = e(x')$ is equivalent to $\exists x' \, P(g(x)) \wedge x' = g(x)$, which reduces to $P(g(x))$. For example, if the assignment statement "$x := x + 1$" has a precondition P equal to $(x = 1)$, then $g(x) = x - 1$ and the postcondition $P(g(x))$ is $(x - 1) = 1$, or $x = 2$, as we would expect.

For a "noop" statement, the postcondition is just the precondition, since the state does not change.

For a conditional statement S of the form "**if** B **then** S_1 **else** S_2," the precondition for S has to be translated into the preconditions for S_1 and S_2; then, after the postconditions for S_1 and S_2 have been found, they have to be combined to form the postcondition of S. Let P be the precondition of S and P_i be the precondition of S_i. Similarly, let Q be the postcondition of S, and Q_i, the postcon-

dition of S_i. Then

$$P_1 = P \wedge B,\ P_2 = P \wedge \sim B$$

$$Q = Q_1 \vee Q_2.$$

The precondition for S_1 is the same as the precondition for S, with the additional fact that B is known to be true. In the same way, when S_2 is executed, B must be false. The outcome of S is either the outcome of S_1 or the outcome of S_2.

For a sequential statement S of the form "S_1; S_2," the precondition of S_1 is of course the precondition of S; the precondition of S_2 is the postcondition of S_1; and the postcondition of S is the postcondition of S_2.

A loop statement S of the form "**while** B **do** S_L" presents some problems because there are circular dependencies between the predicates. The precondition of the loop body S_L depends on the postcondition of S_L, since control can return to the beginning of the loop from the end. But of course the postcondition of S_L depends on the precondition. Therefore, while we can write relations defining the precondition of the loop, it is not usually possible to solve for the precondition. In practice the user must supply a so-called *loop invariant* or *inductive assertion* P_L that satisfies the following conditions:

(L1) $$P \wedge B \supset P_L$$

(L2) $$Q_L \wedge B \supset P_L$$

where P is the precondition of S, Q is the postcondition of S, and Q_L is the postcondition of S_L when P_L is the precondition. Notice that the loop invariant functions like the inductive invariant in the inductive proofs we did earlier. In fact the conditions ($L1$–$L2$) in effect describe an inductive proof of the loop's behavior, with $L1$ the base step and $L2$ the inductive step. The postcondition of the loop body, combined with the fact that the loop condition is false, becomes the postcondition of the **while** loop:

(L3) $$Q_L \wedge \sim B \supset Q$$

To see how this system works, we will first verify a very simple multiplication algorithm that multiplies A by B using successive addition and leaves the result in *product*.

```
(1)   product := 0;
(2)   if A <> 0 then
(3)       while B < 0 do
              begin
(4)           product := product + A;
(5)           B := B - 1
              end
(6)   else noop
```

Statements are numbered on the left for purposes of identification. We use A_0 and B_0 for the initial values of the input variables A and B, so the precondition to the entire description, which is the precondition to the assignment statement (1), is $A = A_0 \wedge B = B_0$. Call this P_0. The postcondition of the assignment is $P_0 \wedge product = 0$. This becomes the precondition to the conditional (2), which follows it in sequence. The precondition to the **while** loop (3) is $P_0 \wedge product = 0 \wedge A \neq 0$, since it is in the **then** part of the conditional. For the loop invariant, we choose $product = (B_0 - B) * A_0 \wedge A = A_0$. To check that this is a valid invariant, we have to verify conditions L1 and L2. L1 is easily seen to be true, since $product = 0 \wedge B = B_0$ implies that $product = (B_0 - B) * A_0$, with both sides equal to 0. The other part of the invariant, $A = A_0$, is part of the precondition. Using the loop invariant as the precondition to the loop body, and thus to the assignment (4), gives $product = (B_0 - B) * A_0 + A \wedge A = A_0$, which is equivalent to $product = (B_0 - B + 1) * A_0 \wedge A = A_0$. Pushing this through the next assignment (5) and using the rule (AS), we replace B by $B + 1$, giving

$$product = (B_0 - (B + 1) + 1) * A_0 \wedge A = A_0$$

$$\equiv product = (B_0 - B) * A_0 \wedge A = A_0.$$

This is just the loop invariant, proving condition (L2). We use condition (L3) to get the postcondition for the while loop: $product = (B_0 - B) * A_0 \wedge B = 0$, which of course reduces to $product = B_0 * A_0$. Now on the **else** side of the conditional, the precondition to the noop (6) is $P_0 \wedge product = 0 \wedge A = 0$, which is also its postcondition. This reduces to $product = 0 \wedge A_0 = 0$. Finally the postcondition of the conditional, which is the postcondition for the description, is $(product = 0) \wedge (A_0 = 0) \vee (product = B_0 * A_0)$, which implies $product = B_0 * A_0$. Therefore, the description meets the specification of a multiplier.

As simple as it is, this example illustrates the basic method used for proving that a hardware description meets a behavioral specification. A precondition is constructed, giving symbolic initial values to the input variables, and the behavioral specification is formulated as a postcondition on the output variables, relating them to the initial input values. An invariant must be chosen for each loop. Then the preconditions are propagated through the description, the loop invariants are verified and propagated, and the resulting postcondition is checked to see that it implies the specified postcondition. This is often called *symbolic simulation*, because it amounts to "executing" the description while using symbolic values such as A_0 for the inputs instead of actual numbers or bit patterns.

This basic method works when the system simply takes in its inputs, computes, and gives its outputs. As we have already observed, however, hardware typically takes a continuous stream of inputs and maps it into a stream of outputs. It is, therefore, necessary to extend the predicate transformer method to handle that kind of situation. One way to do this, which was originally used for programs by Owicki and Gries [65], is to annotate the description with extra variables that function like time variables, keeping track of where the system is in its sequences of inputs and outputs.

To illustrate this method, we return to our example of

a series-to-parallel converter. First we give a description of the converter in the simple hardware description language we have been using:

```
(1)   while not start[i] do
(2)       noop;
(3)   ready := false;
(4)   count := 8;
(5)   {i := 0}
(6)   while (count <> 0) and not start[i] do
          begin
(7)       {i := i + 1}
(8)       shiftreg := shiftreg • 2 + in[i];
(9)       count := count - 1;
          end;
(10)  out := shiftreg;
(11)  ready := true
```

Here we model the input port "in" as a sequence or array of bits. We also add the variable i as a sequence counter. Since i is a construct used only for the verification, the statements manipulating i are in curly brackets. *start* is also modeled as a sequence. To show that the cycle only begins when *start* is set, we begin with a wait loop on *start*. We also show that setting *start* during the main loop will abort the computation. The verification only works if *start* is *true* on the first cycle ($i = 0$) and *false* for the remaining cycles (1 through 8). Therefore, we include in the initial conditions the predicate S_0:

$$S_0 = start[0] \wedge \forall j\ range(j, 1, 8) \supset (\sim start[j]).$$

By going through the initial loop and assignments, we get the precondition for the main loop (6), which is $S_0 \wedge \sim ready \wedge (count = 8) \wedge (i = 0)$. For the loop invariant, we use

$$(P_L)\quad S_0 \wedge \sim ready \wedge (count = 8 - i) \wedge (i < 8) \wedge shiftreg[1:i] = \sum_{k=0}^{i-1} in[i-k] * 2^k.$$

Checking that this follows from the precondition, we observe that the first two conditions are present in the precondition in exactly the same form, the third and fourth follow from the fact that $count = 8$ and $i = 0$, and the last is trivially true, since it involves an empty range. Next we propagate P_L through the loop body. The effect of line (7) is to substitute $i - 1$ for i, giving

$$(P_7)\quad S_0 \wedge \sim ready \wedge count = 8 - (i-1) \wedge (i-1) < 8 \wedge shiftreg[1:i-1] = \sum_{k=0}^{i-2} in[i-1-k] * 2^k.$$

To find the effect of the next assignment (8), we replace *shiftreg* by $shiftreg/2 - in[i]$, or, equivalently, we substitute $2 * X + in[i]$ for X in the expression $shiftreg = X$. This gives

$$(P_8)\quad S_0 \wedge \sim ready \wedge count = 8 - (i-1) \wedge (i-1) < 8 \wedge shiftreg[1:i] = 2 * \sum_{k=0}^{i-2} in[i-1-k] * 2^k + in[i].$$

The last term is equivalent to $shiftreg[1:i] = \Sigma_{k=0}^{i-2} in[i-1-k] * 2^{k+1} + in[i] = \Sigma_{k=1}^{i-1} in[i-k] * 2^k + in[i]$ by changing the summation variable k. Since $in[i] = in[i-k] * 2^k$ with $k = 0$, we can combine this with the sum to give $shiftreg[1:i] = \Sigma_{k=0}^{i-1} in[i-k] * 2^k$. For the last assignment in the loop body (9), we substitute $count + 1$ for *count* to give

$$(P_9)\quad S_0 \wedge \sim ready \wedge (count = 8 - i) \wedge (i-1) < 8 \wedge shiftreg[1:i] = \sum_{k=0}^{i-1} in[i-k] * 2^k.$$

This is the postcondition for the loop body. It is the same as the loop invariant P_L except it has the condition $(i - 1) < 8$ instead of $i < 8$. But the condition $count = 8 - i$ combined with the fact that $i \leq 8$ and the loop condition $count \neq 0$ implies that $i < 8$. Thus P_9 and the loop condition imply P_L, satisfying condition (L2) for the invariant.

From S_0 and the fact that $i \leq 8$, $start[i]$ is *false*, so the negation of the loop condition implies $count = 0$. Combining this with Q_L gives the postcondition for the loop:

$$(P_6)\quad S_0 \wedge \sim ready \wedge (i = 8) \wedge shiftreg[1:8] = \sum_{k=0}^{7} in[i-k] * 2^k.$$

When we substitute in the effects of the last two assignments, we get the final postcondition for the description:

$$(P_{11})\quad S_0 \wedge ready \wedge (i = 8) \wedge out[1:8] = \sum_{k=0}^{7} in[8-k] * 2^k.$$

This implies that at the completion of the execution, *ready* is 1, and for k from 1 to 8 $out[k] = in[9-k]$, which is the original specification.

Of course, to be more precise in this example, we should have modeled *ready* as a sequence also. It then would have been possible to show that ready remains at 0 until after the eighth cycle, at which time it is set. We should also have been more careful about the distinction between Booleans and integers. For example, we treated *in* as an integer when it really is just 1 bit. The translation between bits and integers always has to be handled very carefully because there is great potential for confusion and error there. Nevertheless, what we have done is sufficient to show how these techniques can be extended to handle sequential inputs and outputs, and also some of the difficulties involved.

A number of hardware verification systems have been built using these methods [17], [48], [66], [20]. They differ in the languages used to describe the hardware, the logic used for the specifications, the techniques used for manipulating and comparing predicates, and the types of hardware-specific features they can handle. All of them, however, have the same basic approach of deriving a set of predicates to describe the behavior of a hardware description and comparing them to a specification that is expressed in formal logic.

Predicate transformers can also be used to compare hardware descriptions at different levels of detail. This is useful in verifying that a low-level specification, at the microprogram or logic level, for example, correctly implements a specification that is written as a hardware description at a more abstract level. For example, we might want to verify that a circuit made up of a counter and flip-flops tied together as a shift register correctly implemented our serial-to-parallel converter. To do this, we could write a description of the implementation in the same language, or perhaps a lower level one. The description of the implementation would have a different structure from the high-level description. For example, it would have to have an inner loop that shifted the individual bits in the array of flip-flops. There might also be extra internal variables in the implementation that are not present in the more abstract description. The behavior of the two descriptions is compared by deriving the postcondition for both of them and checking that they are equivalent with respect to the input and output variables.

One technique that is often used to simplify the verification is to add "checkpoints" at corresponding places in the two descriptions where the states can be compared. For example, one might find the point in a microprogram where the execution of a certain portion of the machine language instruction has been completed and the registers visible at the architectural level are in a well-defined state. At this point, relations between values in corresponding registers in the two descriptions can be compared. The advantage of adding these checkpoints is that only the paths between adjacent checkpoints need to be checked. If all the paths have the same behavior as expressed by the relation between preconditions and postconditions, then it follows by an inductive argument that the overall behavior is also the same. In particular, if enough checkpoints are added to break all loops, then only cycle-free paths need to be checked. The verification process can thus be much more highly automated, since there is no need to find loop invariants, a process that cannot be mechanized.

This approach has often been used in hardware verification, including the verification of microcode [18], [8], [9], a 10 000-transistor signal-processing chip [63], the verification of logic implementations [15], [19], and even the verification of layout [44]. These systems and others like them differ in input languages, the type of constructs handled, and the way the predicates are represented and manipulated. The basic approach is the same, however. Symbolic simulation is used to derive logic expressions describing the states in two parallel hardware descriptions, and these are checked for correspondence using some form of automatic theorem prover.

Other groups have combined temporal logic with symbolic simulation. One example of this is the DDL Verifier of Maruyama and Fujita [47]. This system takes a register-transfer hardware description in the DDL language and checks assertions supplied by the user for validity with respect to that description. The logic used is based on first-order temporal logic with equality, but it is still restricted enough that the proofs of individual assertions can be automated and done efficiently.

A different approach to combining static logic and dynamic behavior has been developed by McFarland and Parker [50]. They use a method similar to symbolic simulation to derive predicates describing the relation between input and output values for a system described in a simple hardware description language. Then they imbed these in expressions much like regular expressions, a formalism not unlike predicate path expressions [1], to show the dynamic relationships between the sequences of input/output events. This avoids the rather awkward practice of adding extra variables and structures to a description in order to be able to describe sequences of inputs and outputs. The formalism, called behavior expressions, has been proved complete and consistent with respect to an operational definition of hardware execution, and it has been used to prove that certain optimizing transformations used in high-level hardware synthesis are behavior-preserving. More recently, behavior expressions have been used to analyze part of an existing high-level synthesis system, exposing several significant errors [49].

5.2. *Finite-State Machines*

In theory, any digital system can be modeled as a finite-state machine, and there is a well-developed theory for analyzing such models, including checking their equivalence. It would seem, therefore, that finite-state-machine theory would provide a good basis for verification.

A finite-state machine M is defined by a tuple $(I, O, Q, Q_0, \delta, \lambda)$, where I is a set of inputs, O a set of outputs, and Q a set of states, with a set of one or more initial states Q_0. δ is the state transition function that maps the present state and input to the next state, and λ is the output function that maps the present state and possibly the input to the current output. A finite-state machine can be viewed as a *transducer*, producing a sequence of outputs for each possible sequence of inputs. In this view, two machines are equivalent if they produce the same output sequence for every possible input sequence.

It is sometimes useful to view a finite-state machine as an *acceptor* or *recognizer*. This is done by ignoring the output function and defining a set Q_F of one or more *final states*. A machine M is said to *accept* a string S, where S is a sequence of elements from its input set I, if, when M is started in an initial state and is given the sequence of

elements from S as inputs, it finishes in a final state $f \in Q_F$. The set of strings accepted by M is called the *language* accepted by M. In this view, two machines are equivalent if they accept the same language.

A pair of machines with the same input set, $M_1 = (I, O_1, Q_1, Q_{0,1}, \delta_1, \lambda_1)$ and $M_2 = (I, O_2, Q_2, Q_{0,2}, \delta_2, \lambda_2)$ can be *composed* to form a single machine $M = (I, O_1 \times O_2, Q_1 \times Q_2, Q_{0,1} \times Q_{0,2}, \delta, \lambda)$ which is made up of the two machines running in parallel. The states of M are pairs of states, one from M_1 and one from M_2. The state-transition function δ of M is defined to map pairs of states to pairs of states by applying δ_1 to the first state in the pair and δ_2 to the second one. In other words,

$$\delta((q_1, q_2), i) = (\delta_1(q_1, i), \delta_2(q_2, i)).$$

The output function is defined in much the same way. If M_1 and M_2 are defined to have final states $Q_{F,1}$ and $Q_{F,2}$, the final states of M are all those pairs (q_1, q_2), where q_1 is a final state of M_1 and q_2 is a final state of M_2. M is called the *product machine* of M_1 and M_2, written $M_1 \otimes M_2$.

Parallel composition gives one method for finding the equivalence of two acceptor machines M_1 and M_2. First, each of the machines M_i is complemented to find $\overline{M}_i$, the machine that accepts just those strings that M_i does not accept. Then M_1 is equivalent to M_2 if and only if $M_1 \otimes \overline{M}_2 = \varnothing$ and $M_2 \otimes \overline{M}_1 = \varnothing$. This is true because the language accepted by the product of two machines is just the intersection of the languages accepted by the individual machines, and the machines accept the same language when there is no string accepted by both M_1 and the complement of M_2 and *vice versa*.

One way of checking if the language accepted by a state machine is empty is to do a reachability analysis. This normally involves building a state-transition graph for the machine. In a state-transition graph, each state is represented by a node, and each transition $s_i \rightarrow s_j$ as an arc from the node for state s_i to the node for state s_j. The transition graph is built by starting with the initial states and determining all the transitions that can be made from those states on any valid input. Those transitions are added as edges in the graph. Any time a new state is found as the destination of a transition, a node for that state is created in the graph, and all the transitions from that state are added to the graph. This continues until no new states are found. The advantage of this procedure is that only those states and transitions that can be reached from the initial states by a valid sequence of inputs are recorded in the graph. The number of reachable states and transitions may be considerably less than the total number of possible states and transitions. This is especially true for a composite machine, where a transition $(s_{1a}, s_{1b}) \rightarrow (s_{2a}, s_{2b})$ is often not valid, even though the transitions $s_{1a} \rightarrow s_{2a}$ and $s_{1b} \rightarrow s_{2b}$ both are, because there does not exist a single input that takes both s_{1a} to s_{1b} and s_{2a} to s_{2b}. The language accepted by a state machine is empty if and only if no final state appears in the graph produced by this reachability analysis.

A straightforward use of finite-state machine theory for verification is to model both the implementation and the specification as finite-state machines and to check them for equivalence as transducers, or to check that the specification machine is contained in the implementation machine, meaning that they give the same outputs on all inputs for which the specification is defined. The latter definition is more realistic because it recognizes that the implementation is more detailed than the specification, and will therefore be defined for cases not considered in the specification.

An example of how state machine theory can support verification is the algorithm developed by Devadas, Ma, and Newton to check containment of finite-state machines [21]. The algorithm takes two state-machine descriptions. One, the specification, is described at the register–transfer level, while the other, the implementation, is described as combinational logic and latches. State-transition tables for the two machines M_S and M_I are derived from the specification and implementation descriptions, respectively. Then a composite machine is formed by taking the product of M_S and M_I. The final states of the product machine are defined to be those states in which the output of M_S and the output of M_I are not equal. The machines are equivalent if the language accepted by the product machine is empty, which is determined by a form of reachability analysis.

To make the procedure more efficient, the algorithm seeks to minimize the size of the extracted machines, especially M_I. It first inspects the specification to find all the valid output patterns, then finds the compliment of that set, $\overline{O}_S$. Any state encoding in the implementation that gives an output in $\overline{O}_S$ is taken to be an invalid state and is eliminated from consideration. The specification is also used to find all of the valid input patterns. When M_I is extracted, only the valid states and inputs are included.

This algorithm allows sequential machines at different levels of abstraction to be checked for equivalence automatically. It also shows some of the limitations of the state-machine model for verification, however. First of all, the whole system must be described as a state machine. This is appropriate for small to medium size controllers, but it breaks down when the system has appreciable memory. For example, an n-bit register or counter has 2^n states, all of which are reachable. Even a very efficient algorithm like the one described above can only handle a few hundred states. Second, the specification can only be expressed as a state machine, even if the description can be written in a register–transfer language. Sometimes that is acceptable, but other times it is desirable to give the specification at a more general level, either as a set of input–output relations or as a set of properties that the implementation must satisfy. Finally, in the state-machine formulation, the specification and implementation must correspond closely. Even if it is not necessary to identify internal states, it is still required that the inputs, outputs, and transitions be identical. For example, in the methodology described above, the specification must be

written in terms of individual clock cycles that correspond exactly to the state transitions of the implementation. This precludes using a specification that describes behavior in terms of macrocycles, where each macrocycle may correspond to several clock cycles in the underlying machine.

One way of attacking the complexity of searching large state spaces is to treat sets of states symbolically. This is sometimes called *implicit* state enumeration. In a machine with n Boolean state variables, each possible state is designated by a unique n-bit code. Any set of states, therefore, can be represented by a set of n-bit Boolean vectors or, equivalently, a *characteristic function* χ: $2^n \rightarrow \{0, 1\}$, which is 1 on those codes representing states in the set and 0 on all others. Searching and analyzing state spaces, therefore, reduces to a problem of manipulating Boolean functions, which means that the techniques that have been developed for dealing with Boolean functions efficiently are now available for state-machine analysis. For example, the characteristic function can be represented by a BDD or similar structure, which is often much more compact than an explicit list of all the states. Moreover, set operations can be represented by Boolean operations on the BDD's; e.g., the union of two sets is found by ORing their BDD's. To see how this is used for state-space search, suppose a finite-state machine has n state variables, so that each state is considered to be an element of 2^n. Let $X = 2^m$ be the set of possible inputs to the machine. Let F: $2^n \times 2^m \rightarrow 2^n$ be the Boolean function that represents the transition function δ of the machine; and let $[f_1, \cdots, f_n]$ be the component function of F. Then, for any set A of states in the machine with characteristic function χ_A, χ, the characteristic function of the set of states reachable from A by one transition of the machine can be computed by:

$$\chi(y_1, \cdots, y_n) = \exists s\, \exists x\, \chi_A(s) \wedge \left(\bigwedge_{j=1}^{n} y_j \equiv f_j(s, x) \right). \tag{C1}$$

Equation C1 is iterated, starting with χ_I, the characteristic function of the initial states, until it converges, with χ then equal to the characteristic function of all states reachable from the initial states.

The number of steps involved in the computation of C1 using BDD's or similar representations and the size of the BDD's can still be exponential in the number of state variables n. The number of iterations needed to reach convergence can also be exponential. Nevertheless, a number of heuristics have been developed that seem to work reasonably well on examples with 50 or more state variables [16].

The COSPAN system [34] is a state-machine verification system that addresses that last two of the problems listed above. COSPAN models a system as a set of interacting processes, where each process is described by means of its state variables and state transitions. Each process also has an output called a "selection." At each step of its execution, a process chooses a value for its selection and moves to the next state. The choice of selection may be nondeterministic, but once it is chosen, the state transition is deterministic. This makes analysis of the description much more efficient. The state-transition function and the conditions for a state transition may be expressed symbolically, rather than enumerated as in a normal state-machine description. For example, it is possible to express directly the fact that if the state s of a process satisfies the condition $s < 10$, the next state will be $s + 1$.

The specification for a system in COSPAN is also described by means of finite-state processes, but in a different way from the usual state-machine formulation. The designer usually writes one or more tasks that monitor the behavior of the implementation model, checking that it has certain required properties. For example, a task to check for mutual exclusion in the use of a shared buffer might be set to go into a special error state if two processes gain access to the buffer at the same time.

In COSPAN the implementation model and the tasks are described by finite-state automata called L-automata. L-automata are similar to Buechi automata and Muller automata, in that they accept infinite strings, either by cycling in a set of selected states forever, or by repeating a set of selected transitions infinitely often. For example, the following is a task used to monitor the behavior of a model of a series-to-parallel converter similar to the one discussed in Section III.

```
monitor TASK
 import shift, ctr, watch
 stvar $: (checking, OK, BAD)
 recur checking->OK
 init checking
 trans
 checking
   ->OK  :(ctr.#=1) * (shift.# = watch.#)
   ->BAD :(ctr.#=1) *~(shift.# = watch.#)
   ->$   :else;
 OK
   ->checking  :true;
 BAD
   ->BAD  :true
end
```

The model has two components, a shift register (*shift*) and a counter (*ctr*). The TASK monitor imports the output of both so that it can use them as conditions for its state transitions. It also imports the output of *watch*, an auxiliary process that collects the data inputs and assembles them into a parallel word. TASK has three states for its state variable ($). The state transitions are given in the "trans" section. For each present state, the possible state transitions are listed in the form

→ next state :condition

where "condition" is a Boolean expression giving the conditions under which that transition takes place. TASK

remains in the "checking" state until the counter output (ctr.#) becomes 1, which occurs when the counter reaches 0. When the counter output becomes 1, TASK checks the output of *shift* against the output of *watch*. If they are equal, it goes into the "OK" state and returns to the "checking" state on the next cycle. If they are not, it drops into the "BAD" state, where it remains. The "recur" statement says that the behavior of the system is acceptable as long as the "checking" to "OK" transition occurs infinitely often. This means that the counter always resets and counts properly, and that when each set of 8 bits is assembled, the output of the shift register has the proper value.

COSPAN is not a simulator. It checks a specification by building the state-transition graph of the product of all the processes and tasks in the specification and checking whether there are any bad cycles, that is, cycles that do not include either edges or sets of states that are defined to be accepting. COSPAN has very efficient procedures for building and analyzing the state-transition graph [42], so that it can easily handle many thousands of states. Recently, it has been made even more efficient by adding heuristics for the implicit enumeration of the state space [72]. This is fortunate, because it is subject to the same problems of state-space size that face other state-machine analyzers.

COSPAN supports hierarchical development of systems. A system is first described at a high level of abstraction, and certain global properties are verified. Then the system, or certain components of it, are developed in greater detail, and properties of the detailed designs are verified. At each step it must be shown that replacing the more abstract description by a more refined one does not invalidate the earlier proofs. This methodology has been used in the development of substantial systems, both hardware [27] and software [34].

Reachability analysis and equivalence checking have been applied to several other models that are similar to or based on state machines, including traces [22], Petri nets [60], [40], [28], and state charts [35].

5.3. Model Checking

One example of a practical verification system that combines temporal logic with the idea of representing a circuit as a computational structure and attaching predicates to various points in that structure is the extended model checker (EMC) of Clarke *et al.* [4], [23].

The EMC represents circuits as labeled state-transition graphs. This is like a state-transition graph for a state machine, except that the node for each state is labeled with propositions that are determined to be true in that state.

There are two ways of specifying a circuit and turning the specification into a state-transition graph. In the first, a state-machine description can be extracted from a logic or switch-level description of a circuit. This is done by using a simulator to trace the propagation of logic values through the circuit in response to different combinations of inputs. The simulator uses a unit delay model. That is, it assumes that it takes one unit of time for the inputs to a gate to propagate to its outputs. Each distinct assignment of logic values to the nodes in the circuit is recorded as a separate state. The number of states that need to be recorded is minimized by simulating only those input patterns that are specified as feasible by the user.

The state graph can also be compiled from a somewhat higher level description of the circuit. This higher level description is written in a language called SML (state-machine language), a hardware description language with a Pascal-like syntax. SML is oriented toward the description of the state machines, so its basic data type is Booleans. The primitive statements of the language are *raise*(x), which assigns a logical *true* to a Boolean variable x, and *lower*(x), which sets x to *false*. These statements can be embedded in higher level control structures such as conditionals, loops, and parallel execution statements. The semantics of these statements include an implicit model of a clock, so the state transitions implied by an SML description are well-defined. It is, therefore, relatively straightforward to translate on SML description into a state-transition graph.

Once the graph has been obtained by whatever means, the user can enter formulas to be checked for validity by the system. The formulas are written in a variation of temporal logic called computation tree logic (CTL). CTL allows each state in the logic to have several successor states, so that in general there are many paths that emanate from a given state. The temporal operators included in the language reflect this fact. For example, $\mathbf{EXf_1}$ means that f_1 is true in at least one direct successor to the current state, and $\mathbf{A[f_1Uf_2]}$ means that on every path i from the current state, there exists a state S_i where f_2 is true, and in every state on the path from the current state to S_i, f_1 is true. Other operators are defined in terms of the principle operators. For example, $\mathbf{AGf_1}$ means that on every path from the current state, f_1 is true in every state on that path. It is evident that AGf is the CTL equivalent of $\Box f$.

CTL, like temporal logic in general, is particularly useful for expressing properties of controllers, protocol generators, and other interactive systems. For example, the requirement for a bus interface that a *ready* flag stays low until the value a is put on the output line *out*, at which time *ready* is asserted, can be specified as follows:

$$A[\sim ready\ U\ (out = a) \wedge ready].$$

Once the user submits a formula to the EMC, it is checked automatically against the state graph. If the formula is not true, the system gives a counterexample by showing one state or path for which the formula does not hold. The system has a very efficient algorithm for checking formulas. It involves taking a proposed formula and labeling all those states in which the formula holds. For a simple propositional formula such as $\sim a$ or $(ready \wedge \sim out)$, this can be done easily, since each state in the graph is characterized by the variables that are asserted in that state. For a complex temporal formula, the graph is first labeled

with the subformulas. Then a search that visits each state at most once can be used to evaluate the formula. For example, suppose the formula to be evaluated is $A[f_1 \; U \; f_2]$. First, all those states in which f_1 holds are so labeled, and similarly for f_2. Then, to check whether $A[f_1 \; U \; f_2]$ holds at a state S, the algorithm recursively checks whether it holds at all the successors of S. If it does, and f_1 holds in S, or if f_2 holds in S, then the formula is true for state S, and S is labeled accordingly. If the search on any path from S reaches S again without having reached a state in which f_2 holds, then the formula is false for S. Similar rules govern the search for the other temporal operators.

To evaluate a formula, the algorithm must visit each state in the graph at most once for each subformula, plus once for the formula itself. The complexity of the algorithm is therefore $O(FN)$, where F is the number of operators in the formula and N is the number of nodes in the graph. In practice, properties of circuits with many hundreds of states have been checked using this algorithm in a few seconds. The system has been used to verify both synchronous controllers and asynchronous circuits [4].

Recently, a new algorithm has been developed for the EMC, using BDD's to encode the state-transition graph. This extends the size of the state space the system can handle by several orders of magnitude [6].

5.4. *Algebraic Systems*

A more radical approach to formal reasoning about hardware is to use a completely different formal system, one that is specifically adapted to expressing the concepts relevant to the behavior of hardware (and software).

One type of system that does this consists of recursive expressions that represent sequences of events generated by communicating processes. When this is applied to hardware, the processes describe individual components—at some level of abstraction—and the events stand for the signals that pass between the components as well as signals to and from the outside. The expression for a hardware system represents exactly those sequences of events that can occur when the hardware is in operation.

This type of formal system provides a framework for reasoning about the behavior of hardware. For example, one can speak about equivalence between expressions, meaning that they allow exactly the same event sequences; and there are rewriting rules for proving equivalence, just as there are in a formal logic.

The hardware verification system of this type that has received the most exposure is probably the CIRCAL system [55], [57]. A CIRCAL description can be thought of as representing a set of interconnected boxes. Each box represents an independent computing agent or process and has one or more connection points called *ports*. There are primitives for reading and writing signals or messages on these ports, and operators for combining these primitives sequentially, conditionally, and iteratively (through recursion). There are also operators for putting boxes together to make larger boxes. CIRCAL has rules for reasoning about expressions, transforming them, proving equivalence, and so on.

Milne has used CIRCAL, for instance, to prove that a circuit that checks for equality of bit vectors meets its specification [56]. He has also proved that a simple "silicon compiler" that translates gates into networks of transistors is correct [54], meaning that for any gate-level input it always produces an implementation that has the same behavior, within the given model of behavior. There is a program, analogous to an automatic theorem prover, that assists in CIRCAL proofs by automatically taking care of the details of some of the smaller steps [73].

CIRCAL developed out of work done by Milne and Milner on modeling concurrent systems [53]. The Calculus of Communicating Systems, which CIRCAL resembles in many ways, also developed out of this work [58]. The main difference between the two is CIRCAL's explicit handling of nondeterminism. SCCS [59] and the dot calculus [52] are similar frameworks for reasoning about the behavior of concurrent systems.

VI. Conclusion

In this paper we have investigated several systems for reasoning about the correctness of sequential hardware. All use formal logic, or another formal system that resembles a logic, as a framework for stating and proving properties about the behavior of hardware designs. They differ, sometimes radically, in the way they represent some of the complex features of hardware, such as temporal behavior, hierarchical organization, and the relation between structure and function. Most important from the point of view of the user, they differ in the extent to which they trade off generality for ease of use. In this regard, they range all the way from the higher order logic systems, which can express anything but require a great deal of expertise and time to do a nontrivial proof, to a system like the EMC, which is much more specialized in the properties it can prove and the types of circuits it can represent, but can do the proofs automatically.

At present, there are very few practical applications of formal hardware verification, although a few real chips have been subjected to formal verification of one sort or another. The question is, what role will formal verification have in the future of hardware design? It is generally acknowledged that with the complexity of modern VLSI circuits, exhaustive simulation and testing is impossible. Thus there is certainly a need for a method of analysis that can establish general properties about a system, not just the responses to individual stimuli. It would seem, therefore, that formal verification has something important to offer to the modern designer.

We must be clear, however, about just what formal verification does offer. It is not true, for example, and never will be true, that formal verification will be able to "guarantee" the correctness of a circuit, as is sometimes implied [14]. There are both practical and theoretical reasons for this, and they occur at every point in the process shown in Fig. 2. First of all, the theorem prover cannot

be guaranteed always to give a correct answer. In fact, for any logic powerful enough to describe most of the interesting properties of hardware, it may not give any answer at all, because of decidability problems. Furthermore, there is always the possibility of error in the verification process. If the proof is done by hand, human error cannot be ruled out. Given the size and complexity of the proofs that must be done, it is indeed quite likely. On the other hand, if a computer is used to produce the proof, to assist in it, or even just to check it, then the result is only as reliable as the program used. An automated theorem prover or proof checker is far more complex than the hardware we are trying to verify, so we could not guarantee the correctness of the theorem prover, for all the same reasons we cannot verify the hardware with absolute certainty. Even if we could verify the source code for the theorem prover, we would still have to verify the compiler, the assembler, and the hardware they all run on. Many of the same problems arise when we consider the correctness of the translators from the specification and design representation into the logic.

As difficult as they are, however, the translation and comparison of the specification and design are the easy part, because at least they all take place within a well-defined formal system. When we consider the problem of formulating a formal representation of the actual hardware and its environment, the problems are far greater, because the reality and its representation exist in different universes, which are in some sense incomparable. When we write specifications for a system, for example, we work from a conceptual model of what the system is supposed to do, what inputs it will receive, what outputs are required of it, what the timing of events will be, and so on. Even if we succeed in specifying correctly everything in the model, the model itself is necessarily finite and discrete, and cannot possibly capture the infinite reality of the actual environment in which the hardware will operate [70]. We only hope that we have not left out anything important. Often enough, unfortunately, we do. Many computer systems fail in practice, not because they don't meet their specifications, but because the specifications left out some unanticipated circumstances or some unusual coincidence of events, so that when the unexpected occurred, the system was not able to deal with it. This is not necessarily due to sloppiness or stupidity on the part of the designer or to an inadequate design methodology; it is a fundamental characteristic of the design process. In much the same way, our model of the behavior of a design cannot capture the full reality. If we model it at the gate level, we miss the analog characteristics of the devices, as well as subtleties of timing, the distributed effects of interconnect, and so on. Even if we brought everything down to the quantum level, which would of course be impossible for a complex circuit, we would have to make approximations that might overlook important factors. In addition, there are undoubtedly physical effects that quantum mechanics cannot account for.

Nevertheless, if we drop the unrealistic expectation of a guarantee of correctness, formal verification has much to offer. It is a rigorous, disciplined way of analyzing the behavior of a system and testing it against known requirements. Furthermore, it considers the behavior in general, not just specific instances of it. Submitting a design to formal verification can uncover inconsistencies and unexamined assumptions, and it allows one to work through the implications of a design with more precision and in more detail than is possible with informal checking. Moreover, the whole process of writing a formal specification and devising a verification strategy forces one to think about the system in new ways, raises new questions, and gives new insight into how the system works and why. All of this can help catch errors and lead to a sounder design.

The implication of all this is that formal verification should not be seen as a purely mechanical process in which the designer submits a design and a specification, pushes a button, and waits to see if the red light or the green light will go on. Automation undoubtedly has a role to play—an essential one at that—but it is not the whole story. The importance of formal verification is that, by submitting to the discipline and rigor of the verification process, the designer is forced to think through the design process more carefully and thus comes to understand the design better, with new insights into where it needs to be improved. Even those systems that are most fully automated, like the EMC, are meant to be used interactively. The user examines the design and develops a specification incrementally by proposing properties and having the program check them. If the check fails, the program helps highlight the reason for the failure so that the user can go back and rethink the design.

Formal verification can never be counted on to make designs perfect. If properly developed, however, with attention given to finding the most fruitful mode of cooperation between human judgement and automation, it can become an important tool for making designs more reliable and increasing our confidence in them.

References

[1] S. Andler, "Synchronization primitives and the verification of concurrent programs," in *Proc. Second Int. Symp. Op. Syst.*, Oct. 1978.

[2] G. Birtwistle, J. J. Joyce, and M. Gordon, "Verification and implementation of a microprocessor," *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, Eds. Hingham, MA: Kluwer, 1988.

[3] G. V. Bochmann, "Hardware specification with temporal logic: an example," *IEEE Trans. Comp.*, vol. C-31, pp. 223-231, Mar. 1982.

[4] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra, "Automatic verification of sequential circuits using temporal logic," *IEEE Trans. Comp.*, vol. C-35, pp. 1035-1044, Dec. 1986.

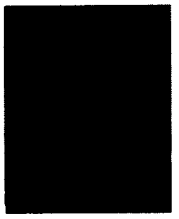
[5] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comp.*, vol. C-35, pp. 677-691, Aug. 1986.

[6] J. R. Burch, "Sequential circuit verification using symbolic model checking," in *ACM/IEEE Proc. 27th Design Autom. Conf.*, pp. 46-51, June 1990.

[7] A. Camilleri, M. Gordon, and T. Melham, "Hardware verification using high-order logic," *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, Ed. New York: North-Holland, 1987.

[8] W. C. Carter, W. H. Joyner, and G. B. Leeman, "Automated experiments in validating microprograms," presented at *Int. Symp. Fault Tolerant Comp.*, June 1975.
[9] W. C. Carter and G. B. Leeman, "Automated proofs of microprogram correctness," presented at *Ninth Ann. Workshop Microprog.*, pp. 51–55, Sept. 1976.
[10] M. S. Chandrasekhar, J. P. Privitera, and K. W. Conradt, "Application of term rewriting techniques to hardware design verification," in *Proc. 24th Design Autom. Conf.*, pp. 277–282, June 1987.
[11] M. Clint and C. A. R. Hoare, "Program proving: jumps and functions," *Act. Inform.*, vol. 1, no. 3, pp. 214–224, 1972.
[12] A. Cohn, "A proof of correctness of the viper microprocessor: the first level," *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, Eds. Hingham, MA: Kluwer, 1987.
[13] A. Cohn, "Correctness properties of the viper block model: the second level," in *Current Trends in Hardware Verification and Automated Theorem Proving*, G. Birtwistle and P. A. Subrahmanyam, Eds. New York: Springer-Verlag, 1989, pp. 1–91.
[14] A. Cohn, "The notion of proof in hardware verification," *J. Automated Reasoning*, vol. 5, pp. 127–139, 1989.
[15] W. E. Cory, "Symbolic simulation for functional verification with ADLIB and SDL," in *Proc. 18th Design Autom. Conf.*, pp. 82–89, 1981.
[16] O. Coudert and J. C. Madre, "Symbolic computation of the valid states of a sequential machine: algorithms and discussion," in *Proc. Int. Workshop Formal Methods VLSI Design*, 1991.
[17] S. Crocker, "State deltas: a formalism for representing segments of computation," Comp. Science Dep., Univ. of California, Los Angeles, CA, UCLA-ENG-7784, Feb. 1978.
[18] S. D. Crocker, L. Marcus, and D. van-Mierop, "Microcode verification project: final report," Inform. Sci. Inst., Univ. of Southern California, Los Angeles, CA, ISI/WP-17, Dec. 1979.
[19] J. Darringer, "The application of program verification techniques to hardware verification," in *Proc. 16th Design Autom. Conf.*, pp. 375–381, June 1979.
[20] S. Dasgupta and A. Wagner, "The use of Hoare logic in the verification of horizontal microprograms," *Int. J. Comp. Infor. Sci.*, vol. 13, no. 6, pp. 461–490, 1984.
[21] S. Devadas, H. T. Ma, and A. R. Newton, "On the verification of sequential machines at different levels of abstraction," in *Proc. 24th Design Autom. Conf.*, pp. 271–276, June 1987.
[22] D. L. Dill, "Trace theory for automatic hierarchical verification of speed-independent circuits," in *Advanced Research in VLSI: Proc. 5th MIT Conf.*, 1988.
[23] D. L. Dill and E. M. Clarke, "Automatic verification of asynchronous circuits using temporal logic," in *Chapel Hill Conf. on VLSI*, pp. 127–143, 1985.
[24] R. W. Floyd, "Assigning meaning to programs," in *Proc. Symp. Applied Math. 19*, pp. 19–32, 1967.
[25] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and improvements of Boolean comparison method based on binary decision diagrams," in *IEEE Proc. ICCAD88*, pp. 2–5, Nov. 1988.
[26] S. M. German and Y. Wang, "Formal verification of parameterized hardware designs," in *Proc. IEEE ICCD85*, 1985.
[27] I. Gertner and R. P. Kurshan, "Logical analysis of digital circuits," in *Proc. 6th CHDL*, pp. 47–57, 1987.
[28] C. Girault, C. Chatelain, and S. Haddad, "Specification and properties of a cache coherence protocol model," in *Advances in Petri Nets 1987*, G. Rozenberg, Ed. New York: Springer-Verlag, 1987, pp. 1–20.
[29] M. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," in *Formal Aspects of VLSI Design*, G. Milne and P. A. Subrahmanyam, Eds. New York: North-Holland, 1986.
[30] M. J. C. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF: A Mechanized Logic of Computation*. Berlin, Germany: Springer-Verlag, 1979.
[31] J. Halpern, Z. Manna, and B. Moszkowski, "A hardware semantics based on temporal intervals," in *Proc. 10th Int. Colloq. Automata, Languages, Prog.*, pp. 278–291, 1983.
[32] L. H. Hanes, "Logic design verification using static analysis," Ph.D. dissertation, Elec. Eng. Dep., Univ. of Illinois at Champaign-Urbana, IL, 1983.
[33] F. K. Hanna and N. Daeche, "Specification and verification using higher-order logic," in *Proc. 7th CHDL*, 1985.
[34] Z. Har'El and R. P. Kurshan, "Software for analytical development of communications protocols," *AT&T Tech. J.*, vol. 69, no. 1, pp. 45–59, Jan./Feb. 1990.
[35] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "Statemate: A working environment for the development of complex reactive systems," *IEEE Trans. Software Eng.*, vol. 16, pp. 403–414, Apr. 1990.
[36] J. Herbert, "The application of formal specification and verification to a hardware design," in *Proc. 7th CHDL*, 1985.
[37] C. A. R. Hoare, "An axiomatic basis of computer programming," *CACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.
[38] C. A. R. Hoare, "Proof of correctness of data representations," *Act. Inform.*, vol. 1, no. 4, pp. 271–281, Nov. 1972.
[39] W. A. Hunt, "FM8501: A Verified Microprocessor," Ph.D. dissertation, Univ. of Texas at Austin, TX, Feb. 1986.
[40] K. Jensen, "Coloured Petri nets. A way to describe and analyse real-world systems—without drowning in unnecessary details," in *IEEE Proc. 5th Int. Conf. on Syst. Eng.*, pp. 395–401, 1987.
[41] F. Kroger, *Temporal Logic of Programs*. New York: Springer-Verlag, 1987.
[42] R. P. Kurshan, "Reducibility in analysis of coordination," in *Proc. IIASA Workshop on Discrete Event Systems*, Aug. 1987.
[43] M. E. Leeser, "Reasoning about the function and timing of integrated circuits using interval temporal logic," *IEEE Trans. CAD/ICAS*, vol. 8, pp. 1233–45, Dec. 1989.
[44] J. Madre and J. Billon, "Proving circuit correctness using formal comparison between expected and extracted behavior," in *Proc. 25th Design Autom. Conf.*, pp. 205–210, June 1988.
[45] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *Proc. ICCAD88*, pp. 6–9, Nov. 1988.
[46] Z. Manna, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.
[47] F. Maruyama and M. Fujita, "Hardware verification," *Computer*, vol. 18, no. 2, pp. 22–32, Feb. 1985.
[48] F. Maruyama, T. Uchara, N. Kawato, and T. Saito, "A verification technique for hardware designs," in *Proc. 19th Design Autom. Conf.*, pp. 832–41, 1982.
[49] M. C. McFarland, "A practical application of verification to high-level synthesis," in *Proc. Int. Workshop on Formal Methods in VLSI*, 1991.
[50] M. C. McFarland and A. C. Parker, "An abstract model of behavior for hardware descriptions," *IEEE Trans. Comp.*, vol. C-32, pp. 621–36, July 1983.
[51] T. F. Melham, "Abstraction mechanisms for hardware verification," *VLSI Specifications, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, Eds. Hingham, MA: Kluwer, 1988.
[52] G. Milne, "Abstraction and nondeterminism in concurrent systems," in *IEEE Proc. 3rd Int. Conf. on Distributed Comp. Syst.*, 1982.
[53] G. Milne and R. Milner, "Concurrent processes and their syntax," *J. ACM*, vol. 26, no. 2, pp. 302–321, Apr. 1979.
[54] G. J. Milne, "The correctness of a simple silicon compiler," in *Proc. 6th CHDL*, pp. 1–12, 1983.
[55] G. J. Milne, "A model for hardware description and verification," in *Proc. 21st Design Autom. Conf.*, pp. 251–257, 1984.
[56] G. J. Milne, "Towards verifiably correct VLSI designs," *Formal Aspects of VLSI Designs*, G. J. Milne and P. A. Subrahmanyam, Eds. New York: North-Holland, 1985, pp. 1–22.
[57] G. J. Milne, "CIRCAL and the Representation of Communication, Concurrency and Time," *ACM TOPLAS*, vol. 7, no. 2, pp. 270–298, Apr. 1985.
[58] R. Milner, *A Calculus of Communicating Systems, 92, LNCS*. Berlin, W. Germany: Springer-Verlag, 1980.
[59] R. Milner, "Calculi for synchrony and asynchrony," *Theoretical Comp. Science*, vol. 25, no. 3, pp. 267–310, 1983.
[60] M. K. Molloy, "Descrete time stochastic Petri nets," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 417–423, Apr. 1985.
[61] B. Moszkowski, "Reasoning about digital circuits," Stanford Univ., Stanford, CA, STAN-CS-83-970, 1983.
[62] B. Moszkowski, "A temporal logic for multilevel reasoning about hardware," *Computer*, vol. 18, pp. 10–19, Feb. 1985.
[63] P. Narendran and J. Stillman, "Formal verification of the Sobel image processing chip," in *Proc. 25th Design Autom. Conf.*, pp. 211–217, June 1988.
[64] N. J. Nilsson, *Problem-Solving Methods of Artificial Intelligence*. New York: McGraw-Hill, 1971.

[65] S. Owicki and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *CACM*, vol. 19, no. 5, pp. 279–285, May 1976.
[66] V. Pitchumani and E. P. Stabler, "An inductive assertion method for register transfer level design verification," *IEEE Trans. Comp.*, vol. C-32, pp. 1073–1080, Dec. 1983.
[67] A. Pnueli, "The temporal logic of concurrent programs," *Theoretical Comp. Science*, vol. 13, no. 1, pp. 415–60, Jan. 1981.
[68] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *JACM*, vol. 12, no. 1, pp. 23–41, 1965.
[69] J. P. Roth, "Hardware verification," *IEEE Trans. on Comp.*, vol. C-26, pp. 1292–1294, Dec. 1977.
[70] B. C. Smith, "Limits of correctness in computers," Ctr. Study Lang., Infor., CSLI-85-36, 1985.
[71] V. Stavridou, H. Barringer, and D. A. Edwards, "Formal specification and verification of hardware: a comparative case study," in *Proc. 25th Design Autom. Conf.*, pp. 197–204, 1988.
[72] H. Touati, R. K. Brayton, and R. Kurshan, "Testing language containment for w-automata using BDD's," in *Proc. Int. Workshop on Formal Methods in VLSI Design*, 1991.
[73] N. Traub, "A lisp based CIRCAL environment," Univ. of Edinburgh Dep. Comp. Science, Edinburgh, UK, CSR-152-83, 1983.
[74] T. J. Wagner, "Hardware verification," Stanford Artificial Intelligence Lab., Stanford Univ., Stanford, CA, AIM 304, Sept. 1977.
[75] A. J. Wojcik, "Formal design verification of digital systems," in *Proc. 20th Design Autom. Conf.*, pp. 228–234, June 1983.
[76] M. Yoeli, Ed., *Formal Verification of Hardware Design*. Los Alamitos, CA: IEEE Comp. Soc., 1991.



Michael C. McFarland (S'77–M'85) received the A.B. degree in physics from Cornell University in 1969 and the M.S. and Ph.D. degrees in electrical engineering from Carnegie Mellon University in 1978 and 1981, respectively. He also has M.Div. and Th.M. degrees from Weston School of Theology.

He is an Associate Professor of Computer Science at Boston College, Chestnut Hill, MA, and a consultant at AT&T Bell Laboratories, Murray Hill, NJ. His research interests include high-level synthesis of VLSI designs, the specification and formal verification of digital systems, and ethics in computer science and engineering.

Dr. McFarland is a member of ACM and Computer Professionals for Social Responsibility.