# A Methodology for Hardware Verification Based on Logic Simulation[*]

Randal E. Bryant
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

1991

**Abstract**

A logic simulator can prove the correctness of a digital circuit if it can be shown that only circuits fulfilling the system specification will produce a particular response to a sequence of simulation commands. This style of verification has advantages over other proof methods in being readily automated and requiring less attention on the part of the user to the low-level details of the design. It has advantages over other approaches to simulation in providing more reliable results, often at a comparable cost.

This paper presents the theoretical foundations of several related approaches to circuit verification based on logic simulation. These approaches exploit the three-valued modeling capability found in most logic simulators, where the third value $X$ indicates a signal with unknown digital value. Although the circuit verification problem is NP-hard as measured in the size of the circuit description, several techniques can reduce the simulation complexity to a manageable level for many practical circuits.

## 1. Introduction

Logic simulators provide a valuable tool for testing the correctness of digital circuits. Typically, however, only a limited set of test cases is simulated, and the circuit is presumed correct if the simulator yields the expected results for all cases. Unfortunately, this form of simulation leaves the designer uncertain whether all circuit design errors have been eliminated. Stories abound of errors that remain undetected despite many hours of simulation and even actual circuit operation. Conventional wisdom holds that logic simulators are incapable of

[*]This research was supported by the Defense Advanced Research Projects Agency, ARPA Order Number 4976.

more rigorous verification. They are viewed in the same class as program debuggers—useful tools for informal testing, but nothing more.

Formal verification involves proving that, under some abstract model of system operation, the circuit will behave as specified for all possible input sequences. A formal proof gives strong confidence that the circuit will function correctly. In this paper, we will show that a logic simulator can form the basis of a formal verifier. At first, this claim might seem both obvious and of little practical value, since most systems are too complex to simulate exhaustively. We argue to the contrary on both points. When the circuit has potential for sequential behavior, even simulating all possible input patterns may fail to detect an error. Furthermore, verification by simulation can be made practical for a significant class of circuits.

Formal verification does not guarantee that the actual circuit will operate properly. The assumptions made in the abstract model may not hold in the physical implementation. For example, most methods of verifying digital systems assume that the circuit adheres to a logic abstraction whereby all signals can be represented by discrete values. Without such an abstraction, verification would be tedious, if not impossible. Design errors that cause marginal, nondigital circuit behavior may not be detected by verification against such a model. Similar problems arise in program verification. For example, most proofs of program correctness abstract the finite arithmetic implemented by computers as operations over integers and real numbers. A verification against such a model cannot detect errors due to arithmetic overflow or underflow. In discussing formal verification, we must remember that the level of confidence it provides is only as strong as the degree to which the abstract model matches actual system operation.

## 1.1. Structural Approaches

Most hardware verification methodologies [1, 2, 18, 19, 23, 24, 25, 26] utilize *structural* techniques. In such an approach, the circuit is described hierarchically, where a component is defined at one level in the hierarchy as an interconnection of components defined at lower levels. The system specification consists of behavioral descriptions of the components at all levels in the hierarchy. Verification then involves proving that each component fulfills its part of the specification, assuming that its constituent components fulfill their specifications.

Structural verifiers have several noteworthy strengths. They can exploit the circuit hierarchy to reduce proof complexity, since a proof is required only for each unique circuit component. Many large, but highly structured circuits have been verified structurally. Second, they can be extended to parameterized circuit descriptions, proving the correctness of entire families of circuits [10]. Finally, structural verifiers can apply different modeling abstractions according to the level in the hierarchy, such as representing signals at lower levels as bits and at higher levels as integers [2].

On the other hand, these verifiers have several shortcomings. Even when automated, they require the user to specify the intended behavior of each component in the circuit hierarchy.

The verifier serves largely as a "proof checker", making sure that each component fulfills its specification. Many circuits are not designed to facilitate component specifications, and hence verification requires much tedious effort on the part of the user. Consider, for example, an adder circuit that utilizes carry-lookahead. Although the desired input-output behavior is straightforward to specify, the low level details of the implementation are complex. Furthermore, the circuit contains many different component types and hence requires a lengthy specification and verification. For such a circuit, a verification method that allows the user to deal with the overall input-output behavior would be far preferable.

As a second shortcoming, most structural verifiers use highly simplified models of electrical and timing behavior to make the proof and component specifications tractable. Most assume, for instance, that the circuit components operate as unidirectional logic elements computing outputs in response to their inputs. In actual circuits electrical operation can be far more subtle, so that the behavior of a component depends on its operating environment. As an example, the direction of information flow through a CMOS transmission gate is determined solely by the driving capabilities of the circuitry at either end [11]. Clearly, any specification of such a gate must include restrictions on the environment in which it is placed. As a notable exception to these highly simplified models, Weise [25, 26] has developed a verifier that proves the correctness of MOS circuits under a model that includes detailed electrical and timing information. His verifier automatically checks every environment in which components are placed for compliance with the preconditions for correct operation. In general, however, prospects do not look good for automating structural verifiers to the point where circuits can be verified with little manual effort and with realistic circuit models. Formulating the proper set of assertions about each component requires a more sophisticated reasoning capability than will be automated in the near future.

## 1.2. A Behavioral Approach

This paper proposes a *behavioral* approach to circuit verification. In this approach, the verifier applies logic simulation to compute the circuit response to a series of stimuli chosen to detect all possible design errors. The user is freed from the tedium of proving the correctness of every component. Instead, the circuit is viewed at a higher level in terms of its desired input-output or state transition behavior. More realistic circuit models can be used, because only the simulator need be concerned with the modeling details.

Although this approach to hardware verification overcomes several weaknesses of structural verifiers, it cannot match some of their strengths. Simulation cannot exploit hierarchy very effectively, because the different instances of a component can have different stimuli and hence must all be evaluated. There is also no known way to simulate an entire class of circuits in a single run. Perhaps the ideal verifier would combine both styles. A hybrid approach would use behavioral verification to prove the correctness of a set of components forming some intermediate level in the circuit hierarchy. This would avoid the need to specify the behavior of the low level components and could employ the detailed circuit models required at these levels. Structural verification would then be applied to the hierarchical composition of the intermediate components, exploiting the regularity of their interconnections and their higher abstraction levels. Thus the work presented here should be viewed as complementing structural verification, rather than seeking to replace it.

## 1.3. Overview of the Methodology

The task of evaluating a circuit by simulating its response to a set of stimuli relates closely to the "machine identification" problem first described by Moore [20]. He showed that, in general, no finite set of stimuli could fully characterize the behavior of a sequential system. He suggested overcoming this problem by fixing an upper bound on the total number of system states. Unfortunately, for circuits of significant size, this bound is too high for Moore's identification algorithm to be practical. Instead, our method overcomes the identification problem by simulating the behavior over a three-valued domain with conventional Boolean values 0 and 1, plus a value $X$ representing an undefined or uninitialized signal. Such a capability is found in most logic simulators [14]. In addition to supplying input patterns during simulation, we assume that the user can issue Erase commands, causing all state variables to be set to $X$. Although the power of three-valued simulation has been studied extensively in the context of hazard detection [7, 17, 27], its potential role in circuit verification has not been widely recognized.

In the interest of generality and simplicity, the paper views hardware specification, digital circuits, and logic simulation in rather abstract ways. The desired behavior is specified by a (Moore model) finite state automaton. The circuit is also abstracted as a finite state automaton, with a particular encoding of the states. Although this is an unconventional view of a circuit, we will argue its appropriateness for behavioral verification, where the focus is on how the circuit operates rather than how it is constructed. Circuit verification involves proving that the specification and circuit automata have equivalent input-output behavior. The simulator models the behavior of the circuit automaton, computing new state and output values in response to inputs supplied by the user. A mild monotonicity property is imposed on the simulation of three-valued behavior to capture the notion that $X$ represents an unknown or ambiguous digital value.

It is shown that the style of simulation required to prove correctness must depend on the nature of the system specification. A *definite* system [16, 21], for which the behavior depends on only a bounded number of previous inputs, can be verified by straightforward "black-box" simulation. Black-box simulation involves simply observing the output produced by the simulated circuit in response to a sequence of input and Erase commands with no consideration of the internal circuit structure. Verifying the implementation of an *indefinite* system, on the other hand, requires a more implementation-specific "state transition" simulation. With this method, key circuit state variables are identified, and the different possible state transitions simulated. In either case, the verification requires little, if any, understanding of the detailed circuit design.

Circuit verifiers can err in two different ways. One that rejects a correct circuit gives a *false negative* response, whereas one that accepts an incorrect circuit gives a *false positive* response. This paper is concerned mainly with avoiding false positive responses. Such a response has more potential for danger—it may cause a defective design to be implemented or put into service. Furthermore, deciding whether a simulator has produced a false negative response requires more detailed information about the circuit electronics and the simulation algorithm than can be presented in a general way. However, for the simulation sequences presented in this paper, a false negative response must have a particular form, namely the simulator will produce $X$ on some output when 0 or 1 was expected.

As mentioned earlier, any approach to formal verification guarantees proper circuit operation only if the assumptions made in the abstract model hold in the circuit implementation. For the case of simulation-based verification, we must assume that the abstract circuit model provided by the simulator faithfully captures the behavior of the actual circuit. When a circuit has been "verified" by simulation, it simply means that any further simulation would not uncover any errors. It is important to maintain this perspective on the problem addressed by this paper. It reflects a weakness intrinsic to any approach to verification and not to simulation alone.

## 1.4. Contents of Paper

This paper presents both theoretical and practical aspects of a hardware verification methodology based on multi-valued logic simulation. Section 2 illustrates the key ideas by means of several circuit examples. Section 3 gives a notation and mathematical background for describing system specifications, logic simulation, and the verification problem. Section 4 gives a formal characterization of the capabilities and limitations of black-box simulation. Section 5 shows how these limitations are overcome by state transition simulation.

Section 6 discusses methods to improve the computational efficiency of the verifier. Two methods are proposed to reduce the computational effort in verifying large digital systems: input weakening and symbolic simulation. Input weakening involves using the value $X$ on an input to represent "don't care", thereby reducing the total number of patterns that need to be simulated. Symbolic simulation involves augmenting a simulator with a symbolic Boolean manipulator to compute the behavior of the circuit over input patterns containing Boolean variables. Section 7 demonstrates a practical application of the methodology to the verification of a random-access memory. By exploiting input weakening, an $n$-bit memory can be fully verified by simulating $O(n \log n)$ patterns, even though the circuit has $2^n$ possible states. Section 8 concludes the paper with a discussion of the method.

All circuit examples shown in this paper are designed in MOS technology. This choice reflects the historical background of the research as well as the belief that MOS circuits form a particularly difficult class for hardware verification. In particular, state can be stored as dynamic charge on capacitive nodes. Unlike in technologies where state is stored only in feedback loops, in MOS technology, every node can potentially store state. In fact, design errors commonly introduce extraneous state and unintended sequential dependencies. It is a nontrivial task simply to verify that a circuit implements a combinational function, rather than a sequential one.

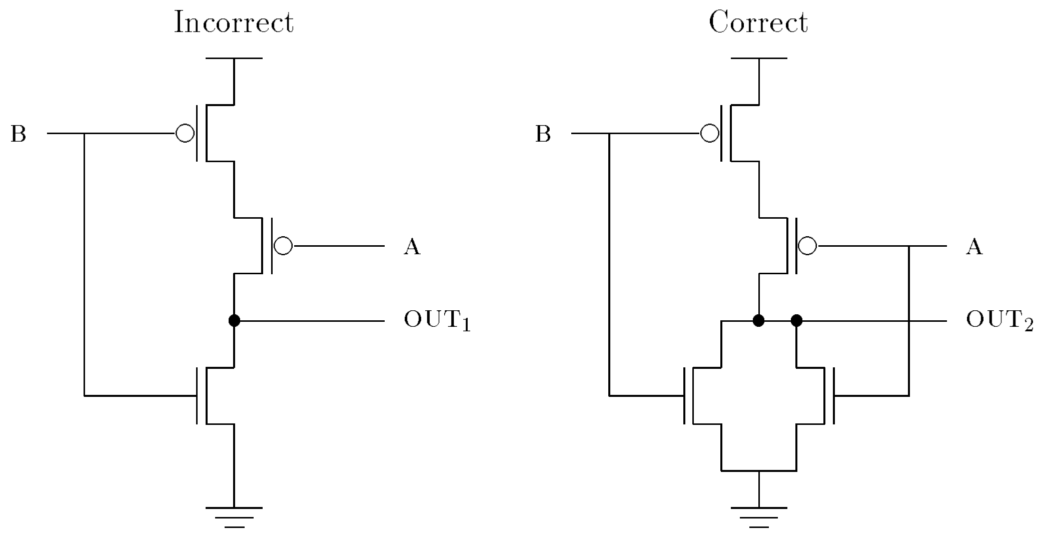

Figure 1: Implementations of a NOR Gate in CMOS

The methodology presented here, however, applies to most digital technologies, with the caveat that circuits are assumed to operate on synchronized input data. Asynchronous systems seems to call for more powerful class of verification tools, such as the model checker of Clarke, *et al* [3, 8], since they cannot be viewed simply as processing a single sequence of input data.

This research provides two major contributions to the state of the art in circuit validation. First, it presents a simple, yet powerful, method of proving the correctness of digital hardware. Second, it provides insights into ways to better utilize a simulator even when only informal validation is sought. It shows that by exploiting a latent capability found in most logic simulators, namely three-valued modeling, more rigorous validation can be obtained at comparable cost.

## 2. Verification Examples

Before proceeding with the mathematical formalism, we present the main concepts via several circuit examples. These examples illustrate several pitfalls of simulation and how three-valued modeling can be exploited to overcome them.

### 2.1. Definite Systems

Consider the seemingly simple task of proving that a circuit implements a NOR logic gate. Figure 1 shows two proposed implementations in CMOS technology [11]. If we were to simulate these circuits using a simulator that can model a MOS circuit at the transistor level [4], the following responses would be produced when the input patterns are applied in the sequence shown:

| A | B | OUT$_1$ | OUT$_2$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |

The two circuits appear identical for all possible input combinations, as they would in actual implementations of the circuits. However, only the second circuit is a valid NOR gate. The first is a two-state sequential circuit, because when A = 1 and B = 0, the output node is electrically isolated from all others and remains charged at its previous value. Due to the order in which the input combinations were applied, it just happened that the previous value of OUT$_1$ equaled the value a NOR gate should produce for this input combination. On the other hand, had this input combination been simulated immediately after the combination A = B = 0, the output would have been 1.

This example illustrates a common problem in testing a circuit by simulation even when rigorous verification is not sought. A design error introduces an unintended sequential dependency in the circuit, but this error remains undetected because of the particular order in which the test sequences are simulated. Clearly, such a condition is not acceptable for formal verification.

Suppose, on the other hand, that an ERASE command is given before simulating each input combination, causing all state variables to be set to $X$. Such a simulation would produce the following results:

| A | B | OUT$_1$ | OUT$_2$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | $X$ | 0 |

The presence of an $X$ on OUT$_1$ indicates that the output of this circuit may not be uniquely defined when A = 1 and B = 0 (or it could be a false negative response by a valid circuit). On the other hand, each input combination produces a unique response for the second circuit, and since the responses match those of a NOR gate, one can conclude the circuit is correct. This conclusion can be drawn without any further information about the structure of the circuit or the number of state variables.

A combinational system such as a NOR gate is 1-definite; its output at any time depends only on the most recent input. The method shown above generalizes to any definite system specification, where the output depends only on the most recent $k$ inputs for some constant $k$. That is, suppose for every possible input sequence of length $k$, setting all state variables

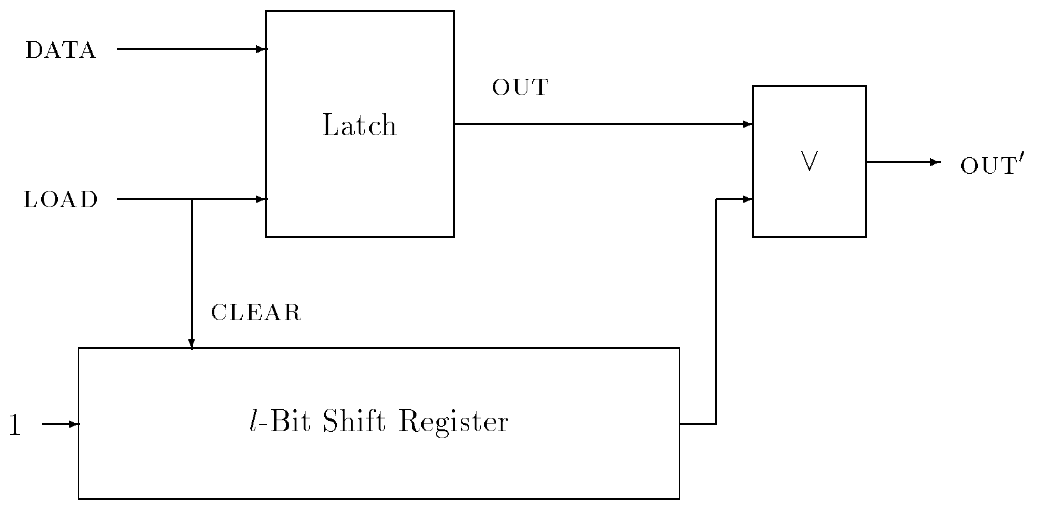

Figure 2: Impostor Latch

to $X$ and then simulating the sequence yields an output equal to the desired value. One can then safely conclude the circuit fulfills the specification.

Observe that this requirement for definiteness applies only to the system specification and not to the circuit being evaluated. For example, the simulator is able to detect the incorrect NOR gate even though the circuit itself is not definite.

## 2.2. Indefinite Systems

Many sequential systems are not definite. For example, a simple 1-bit latch has an output dependent on an input that occurred arbitrarily long in the past as long as no new value is written into it. For such a system, given any value $k$, there will always be an input sequence of length $k$ that does not cause the system to produce a unique output. When simulating this sequence following an ERASE command, even a correctly designed circuit will give an $X$ on the output.

In Section 4 we will show that for any indefinite system specification, there is no way to prove that a circuit fulfills its specification by simply observing the output values resulting from a sequence of input patterns and ERASE commands. This general limitation of black-box simulation can be illustrated using a 1-bit latch circuit, as illustrated in the upper left hand corner of Figure 2. A value $v$ is written into this latch by setting DATA to $v$ and LOAD to 1. This value remains in this latch as long as LOAD is held at 0. Consider a simulation sequence that is claimed to detect any defective latch design. Since the sequence is finite, there must be some value $l$ such that the LOAD input is never held at 0 for $l$ or more consecutive patterns.

Consider the "impostor" circuit of Figure 2 consisting of a correct latch with additional circuitry implementing a "booby trap". Whenever a value is written into the latch, all bits in the booby trap shift register are cleared to 0. The output of the shift register is OR'ed

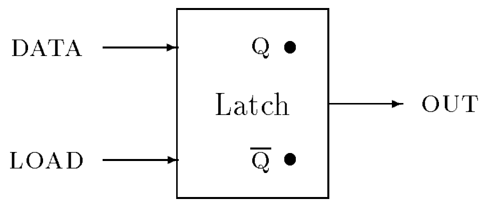

Figure 3: Latch Circuit

with the latch output to produce the circuit output OUT′. Thus, the circuit behaves as a proper latch as long as the shift register output equals 0. If no further data is written, a 1 will shift through the register until it ultimately causes the circuit output OUT′ to equal 1 even if the latched value equals 0. Clearly, this circuit does not behave as a latch should. However, the proposed sequence does not cause enough consecutive shift operations for the impostor circuit to behave differently from the correct one.

Less obviously, even attempts to expose the booby trap by giving ERASE commands or by giving input sequences containing $X$'s will fail to distinguish the correct latch from the impostor circuit of Figure 2. Unlike in the NOR gate example, any action here that would cause the simulator to produce output $X$ for the incorrect circuit would also cause it to produce output $X$ for the good circuit. Thus, the proposed simulation sequence cannot distinguish between a correct circuit and this incorrect one. This argument holds for any simulation sequence by making $l$ sufficiently large.

To verify indefinite systems, more information is required about the circuit state variables and their relation to the states of the system specification. However, in the spirit of black-box simulation, we would like to minimize the amount of detail about the circuit structure that the user must provide. To achieve this goal, assertions about the state variables and how they are transformed by the input values are expressed in a notation similar to the Floyd-Hoare assertion method of program verification [9, 13]. Each assertion is then verified by a short simulation sequence. At the present stage of this research, we require the user to prove manually that the set of assertions forms a complete system specification.

A *circuit assertion*, denoted by an equation of the form *Initial* { *Action* } *Result*, consists of 3 formulas over the circuit input, output, and state variables. Formula *Initial* specifies a precondition on the state variables, *Action* a setting of the input variables, and *Result* a postcondition on the state and output variables. Each formula is a predicate of the form $L_1 \wedge L_2 \wedge \cdots \wedge L_k$, where each $L_i$ is a literal of the form $v = 1$ or $v = 0$, for some circuit variable $v$. A circuit assertion can be interpreted as a statement that, for any circuit state satisfying *Initial* and for any action satisfying *Action*, the resulting circuit state and output will satisfy *Result*.

As an example, consider an implementation of the 1-bit latch illustrated in Figure 3. No internal details of the circuit are shown except that the bit is stored in a feedback path containing two electrical nodes Q and $\overline{\text{Q}}$. The following assertions specify the state transition behavior of the circuit where $a$ ranges over 0 and 1:

$$true \left\{ \text{DATA} = a \ \wedge \ \text{LOAD} = 1 \right\} \text{Q} = a \ \wedge \ \overline{\text{Q}} = \neg a \ \wedge \ \text{OUT} = a$$

$$\text{Q} = a \ \wedge \ \overline{\text{Q}} = \neg a \left\{ \text{LOAD} = 0 \right\} \text{Q} = a \ \wedge \ \overline{\text{Q}} = \neg a \ \wedge \ \text{OUT} = a$$

The first equation asserts that a write operation sets the state of a latch, while the second asserts that the latch state does not change as long as no new data is written. In both cases the value of OUT should equal that of Q.

Given an assertion *Initial* { *Action* } *Result*, its verification by simulation involves the following steps. An ERASE command is given to set all state variables to $X$. For each term $v = a$ in *Initial* (respectively, *Action*), state variable $v$ (resp., input variable $v$) is set to $a$. All input variables not occurring in *Action* are set to $X$. The simulator then computes the resulting output and new state. For each term $v = a$ in *Result*, state or output variable $v$ is tested for equality with $a$. If all of these tests hold, then the assertion is proved.

For example, consider the latch circuit of Figure 3, and the impostor circuit formed by adding the booby trap of Figure 2 to it. Simulating the four sequences specified by the assertions would yield the following results:

| Initial Values | | | | Results | | | |
|---|---|---|---|---|---|---|---|
| Q | $\overline{\text{Q}}$ | DATA | LOAD | Q | $\overline{\text{Q}}$ | OUT | OUT$'$ |
| $X$ | $X$ | 0 | 1 | 0 | 1 | 0 | 0 |
| $X$ | $X$ | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | $X$ | 0 | 0 | 1 | 0 | $X$ |
| 1 | 0 | $X$ | 0 | 1 | 0 | 1 | $X$ |

The impostor circuit passes the first two tests when new data is written, because this causes the shift register to be cleared. For the final two tests, Q and $\overline{\text{Q}}$ are initialized to Boolean values, but all other state variables, including those within the shift register, are initialized to $X$. The shift register output will remain at $X$, causing an $X$ to appear on OUT$'$ and the tests to fail.

## 3. Mathematical Formulation

The examples of the previous section illustrate the main ideas of our verification methodology. We will now proceed with a more formal presentation, showing that these ideas apply to general classes of digital systems. This section develops a mathematical abstraction of logic circuits, simulation, and the verification problem.

## 3.1. Notation

We adopt a notation that represents the system input, output, and state values as vectors. The history of inputs applied to a system is denoted by a sequence of vectors.

A vector with elements $x_1, x_2, \ldots x_n$ is written either $\langle x_1, \ldots, x_n \rangle$ or simply $\vec{x}$. A sequence consisting of vectors $\vec{a}_1, \ldots, \vec{a}_k$ is written $[\vec{a}_1, \ldots, \vec{a}_k]$. The Greek letters $\alpha$, $\beta$, and $\gamma$ are used to denote vector sequences.

| | |
|---|---|
| $p$: | the number of system inputs. |
| $B$: | $\{0, 1\}$, the Boolean domain. |
| $B^n$: | $\{\langle x_1, \ldots, x_n \rangle \mid x_i \in B\}$, Boolean vectors of size $n$. |
| $\mathcal{BI}_l$: | all length $l$ sequences with elements in $B^p$. |
| $\mathcal{BI}$: | $\bigcup_{0 \leq l < \infty} \mathcal{BI}_l$, i.e., all finite sequences with elements in $B^p$. This set represents all possible system input sequences. |
| $T$: | $\{0, 1, X\}$, the ternary domain, partially ordered $X < 1$ and $X < 0$. |
| $T^n$: | $\{\langle x_1, \ldots, x_n \rangle \mid x_i \in T\}$, ternary vectors of size $n$, partially ordered $\vec{x} \leq \vec{y}$ when $x_i \leq y_i$ for all $1 \leq i \leq n$. |
| $\mathcal{TI}_l$: | all length $l$ sequences with elements in $T^p$. |
| $\mathcal{TI}$: | $\bigcup_{0 \leq l < \infty} \mathcal{TI}_l$, i.e., all finite sequences with elements in $T^p$. This set represents all possible simulator input sequences. The set is partially ordered $[\vec{a}_1, \ldots, \vec{a}_s] \leq [\vec{b}_1, \ldots, \vec{b}_t]$ when $s \leq t$ and $\vec{a}_{s-i} \leq \vec{b}_{t-i}$ for all $0 \leq i < s$. |
| $\epsilon$: | the empty sequence. |
| $\alpha\,\vec{x}$: | The sequence consisting of the elements of sequence $\alpha$ followed by the vector $\vec{x}$. |

## 3.2. Information Ordering

The partial ordering $X < 0$ and $X < 1$ orders values by their "information content." That is, $X$ indicates an absence of information while 0 and 1 represent specific, fully-defined values. When speaking of domains ordered by information content, we say that values $a$ and $b$ are "consistent" if either $a \leq b$ or $b \leq a$, and "inconsistent" otherwise. Value $a$ is "weaker" than $b$ if $a < b$, i.e., $a \leq b$ and $a \neq b$.

The information ordering is extended to vectors and vector sequences by adopting the convention that one value is less than another if the elements of the first are consistent with those of the second, but the first contains less information. More precisely, for vectors in $T^n$,

|  |  | $a$ |  |  |
|---|---|---|---|---|
|  |  | 0 | 1 | $X$ |
|  | 0 | 0 | 1 | $X$ |
| $b$ | 1 | 1 | 1 | 1 |
|  | $X$ | $X$ | 1 | $X$ |

|  |  | $a$ |  |  |
|---|---|---|---|---|
|  |  | 0 | 1 | $X$ |
|  | 0 | 0 | 1 | $X$ |
| $b$ | 1 | 1 | 1 | 1 |
|  | $X$ | $X$ | $X$ | $X$ |

|  |  | $a$ |  |  |
|---|---|---|---|---|
|  |  | 0 | 1 | $X$ |
|  | 0 | 0 | 1 | $X$ |
| $b$ | 1 | 1 | 1 | $X$ |
|  | $X$ | $X$ | 1 | $X$ |

|  |  | $a$ |  |  |
|---|---|---|---|---|
|  |  | 0 | 1 | $X$ |
|  | 0 | 0 | 1 | $X$ |
| $b$ | 1 | 1 | 1 | $X$ |
|  | $X$ | $X$ | $X$ | $X$ |

Table 1: All Monotonic Extensions of the OR function

one vector is less than or equal to another if each element of the first is less than or equal to the corresponding element of the second. For sequences $\alpha, \beta \in \mathcal{TI}$, $\alpha$ is less than or equal to $\beta$ if the elements of $\alpha$ are less than or equal to the corresponding final elements of $\beta$. That is, the history given by $\alpha$ is consistent with the most recent history given by $\beta$, but may contain less information. The motive for this convention on the ordering of different length sequences will become clear when we study the monotonicity properties of the simulator. As a special case, sequences of Boolean vectors, $\alpha, \beta \in \mathcal{BI}$, are ordered $\alpha \leq \beta$ when $\alpha$ is a suffix of $\beta$.

For partially ordered sets $D_1$, $D_2$ a *monotonic* function $g \colon D_1 \to D_2$ satisfies

$$a \leq b \Longrightarrow g(a) \leq g(b)$$

for all $a, b \in D_1$. Similarly, a monotonic function over multiple arguments satisfies this property for each argument.

For any program, such as a logic simulator, that processes data ordered by information content, monotonicity expresses an important property. Suppose the program is given a stimulus containing incomplete information, e.g., having some inputs equal to $X$. If the program obeys monotonicity, it will produce a response consistent with, but potentially weaker than, the response it would produce given a stronger stimulus.

For logic simulation, the three-valued behavior of a circuit is generally computed by monotonically extending functions defined over Boolean values to ones defined over ternary values. As an example, the two-input OR function can be extended monotonically from the Boolean to the ternary domain in the four different ways shown in Table 1. These extensions are listed from left to right in order of "pessimism", i.e., by the number different input combinations that are defined to yield $X$. A pessimistic extension is valid, but it will tend to cause false negative responses.

## 3.3. System Specification

The system to be implemented has $p$ inputs and $m$ outputs, each of which may equal 0 or 1. Hence the system may be described as a finite automaton with input alphabet $B^p$ and output alphabet $B^m$.

**Definition 1** *The system* specification *is a triple* $\langle Q, Nspec, Ospec \rangle$ *with*

$Q$: *a finite set of states,*

$Nspec$: *the next state function* $Nspec: Q \times B^p \rightarrow Q$*, and*

$Ospec$: *the output function* $Ospec: Q \rightarrow B^m$*.*

*Function Nspec must be a surjection. That is, for every* $q \in Q$*, there must be a* $q' \in Q$ *and* $\vec{a} \in B^p$ *such that* $q = Nspec(q', \vec{a})$*.*

Requiring *Nspec* to be a surjection implies that every state must be reachable by a transition from some state (possibly itself.) This restriction is imposed for technical reasons but should not limit the class of actual systems under consideration, because there would be no reliable way to put the system into a state that cannot be reached by any state transition. This restriction is milder than the strong connectivity property assumed by Moore [20].

The function *SpecState* is defined to yield the state reached when, starting in some initial state, an input sequence is applied. That is, $SpecState: Q \times \mathcal{BI} \rightarrow Q$ is defined recursively as

$$\begin{aligned} SpecState(q, \epsilon) &= q \\ SpecState(q, \alpha\, \vec{x}) &= Nspec(SpecState(q, \alpha), \vec{x}). \end{aligned}$$

The function $SpecOut: Q \times \mathcal{BI} \rightarrow B^m$ is defined to yield the final output after a sequence of inputs has been applied, i.e.,

$$SpecOut(q, \alpha) = Ospec(SpecState(q, \alpha)).$$

## 3.4. Simulation Model of a Circuit

The simulation of a circuit is also represented as a finite automaton but with states encoded by $s$ ternary variables.

**Definition 2** *A* simulation model *is a triple* $\langle s, Nsim, Osim \rangle$ *with*

$s$: *the number of state variables,*

$Nsim$: *a monotonic next state function* $Nsim: T^s \times T^p \rightarrow T^s$*, and*

$Osim$: *a monotonic output function* $Osim: T^s \rightarrow T^m$*.*

Since the simulator is assumed to faithfully model the actual circuit behavior, we make no attempt to distinguish between the circuit and its simulation model. Thus, we can abstract away many details of the actual circuit. Typically, the automaton modeling a circuit depends on several factors:

- The *logic network* consisting of a set of interconnected elements.
- The *logic model* defining a mapping from networks to logical behavior.
- The *monotonic extension* used to express circuit behavior when some inputs or state variables equal $X$.
- The *clocking methodology* defining the patterns applied to the clock inputs as well as the time points at which data inputs are applied and data outputs are sensed.

Representing the circuit as a finite automaton corresponds to the view typically seen by the user of a logic simulator. The simulation program handles the details abstracted away by our model. It creates data structures representing the logic network and, via its simulation algorithm, implements some monotonic extension of the logic model. The clocking methodology is declared at the outset. From this point on, the user views the simulation task as one of validating a finite automaton. He or she supplies inputs, commands the program to simulate clock cycles (i.e., make state transitions), and observes the outputs. The user may also set and observe the values of internal state variables. Hence, this circuit model is appropriate for the discussion here, since our goal in verification is simply to ensure that the user uncovers any design errors that can be detected by the simulator. Furthermore, by viewing the problem at this level of abstraction, our results apply to a broad class of logic models and simulation programs.

The monotonicity requirement captures the idea that the value $X$ indicates an unknown or ambiguous digital value. When some input or state variable equals $X$, the simulator must compute a new state and output that is consistent with, but potentially weaker than, it would if this input or state variable equaled 0 or 1.

Our formal results hold for any monotonic next state and output functions. To minimize false negative responses, however, we would like the simulator to set an output or state variable to $X$ only when a Boolean value cannot be assigned without violating monotonicity. Unfortunately, such a computation can easily be shown to be NP-hard. Instead, most simulators err on the side of pessimism in the interest of efficiency, with a resultant tendency toward false negative responses. For example, logic gate simulators typically simulate each gate according to its least pessimistic monotonic extension [14].

**Definition 3** *Simulation model* $\langle s, Nsim, Osim \rangle$ fulfills *specification* $\langle Q, Nspec, Ospec \rangle$ *when there exists a relation* $\mathcal{E} \subseteq Q \times T^s$ *(for "encodes") satisfying:*

1. *For every* $q \in Q$ *there exists a* $\vec{z} \in T^s$ *for which* $q \; \mathcal{E} \; \vec{z}$.
2. *For any* $q \in Q$ *and* $\vec{z} \in T^s$
$$q \; \mathcal{E} \; \vec{z} \implies Ospec(q) = Osim(\vec{z}).$$
3. *For any* $q \in Q$, $\vec{z} \in T^s$, *and* $\vec{x} \in B^p$,
$$q \; \mathcal{E} \; \vec{z} \implies Nspec(q, \vec{x}) \; \mathcal{E} \; Nsim(\vec{z}, \vec{x}).$$

By this definition, the simulation model must "cover" the input–output behavior of the specification. That is, for any state of the specification $q$, there must be a circuit state $\vec{z}$ such that, if the two automata were started in their respective states and any input sequence were applied, they would yield identical output sequences. Note, however, that this definition does not place many restrictions on how the model fulfills the specification. There may be circuit states that do not correspond to any specification states, such as those involving invalid combinations of state variables. Neither automata need be reduced—several circuit states may correspond to a single specification state, and vice-versa. There is also no designated initial state of the specification or the circuit.

The above definition of how a simulation model fulfills a specification corresponds closely to the definition by Hartmanis and Stearns of how one sequential machine *realizes* another [12, p. 28]. These authors define realization in terms of a function $\alpha$ mapping each state of the specification machine into a set of states in the realization machine. Mathematically, however, the 2 definitions are equivalent. That is, given an encoding relation $\mathcal{E}$ a mapping function $\alpha$ can be defined as mapping each state $q$ to the set of all states $\vec{z}$ such that $q \mathcal{E} \vec{z}$. Similarly, given a mapping function $\alpha$, an encoding relation can be defined as $q \mathcal{E} \vec{z}$ for every $\vec{z} \in \alpha(q)$. The conditions we impose on $\mathcal{E}$ can be shown are equivalent to those Hartmanis and Stearns impose on $\alpha$.

## 3.5. Simulator

The simulator implements the finite automaton of the simulation model. It maintains a set of values indicating the simulation state $\vec{z} \in T^s$. Five commands are defined for the simulator. For black-box simulation, however, only the first three are allowed.

| | |
|---|---|
| Erase: | causes the simulator to set $z_i$ to $X$ for $1 \leq i \leq s$. |
| Apply($\vec{x}$): | causes the simulator to set $\vec{z}$ to $Nsim(\vec{z}, \vec{x})$. |
| Output: | causes the simulator to print $Osim(\vec{z})$. |
| Set($i, b$): | causes the simulator to set $z_i$ to $b$ for $b \in T$. |
| Observe($i$): | causes the simulator to print $z_i$. |

## 3.6. Simulation Experiment

A *simulation experiment* consists of a finite sequence of simulation commands beginning with Erase, as well as a procedure by which the user decides whether the outcome is acceptable. By "procedure" we mean any method by which the user accepts or rejects a circuit based only on the information provided by the Output and Observe commands in the simulation sequence.

An *effective* experiment for a specification is one that never yields a false positive response. That is, if a simulation model produces an acceptable outcome to the experiment, then it must fulfill the specification

An experiment is *nontrivial* when there is some simulation model $\langle s, \mathit{Nsim}, \mathit{Osim} \rangle$ for which the simulator produces an acceptable outcome. This condition is imposed to eliminate the otherwise effective test of rejecting all circuits.

## 4. Black-Box Simulation

With black-box simulation, the user is limited to the simulation commands ERASE, APPLY, and OUTPUT. No direct observation or modification of the simulator state $\vec{z}$ is permitted. This section identifies the class of systems that can be verified by black-box simulation.

Define the function $\mathit{SimState}: \mathcal{TI} \rightarrow T^s$ as the state of the simulator after giving an ERASE command followed by a series of APPLY commands. More precisely

$$\begin{aligned} \mathit{SimState}(\epsilon) &= X^s \\ \mathit{SimState}(\alpha\,\vec{x}) &= \mathit{Nsim}(\mathit{SimState}(\alpha), \vec{x}) \end{aligned}$$

where $X^s$ denotes a vector of size $s$ with each element equal to $X$.

Similarly, define the function $\mathit{SimOut}: \mathcal{TI} \rightarrow T^m$ as the result that would be printed by an OUTPUT command following the simulation of some input sequence, i.e., $\mathit{SimOut}(\alpha) = \mathit{Osim}(\mathit{SimState}(\alpha))$.

A black-box simulation experiment can be viewed as a criterion for either accepting or rejecting a circuit on the basis of the values of $\mathit{SimOut}(\alpha)$ for some finite number of sequences $\alpha \in \mathcal{TI}$.

### 4.1. Definite Systems

For $k \geq 0$, a specification is *k-definite* when $\mathit{SpecOut}(q_1, \alpha) = \mathit{SpecOut}(q_2, \alpha)$ for any $\alpha \in \mathcal{BI}_k$ and any $q_1, q_2 \in Q$. A specification is *definite* when it is $k$-definite for some $k$. Otherwise it is *indefinite*.

This class of sequential systems was first identified by Kleene [15]. Since that time, various definitions have appeared, viewing sequential systems either as recognizers [21] or transducers [16]. Our definition most closely matches that of Kohavi [16]. However, he defines a $k$-definite system as one for which any input sequence of length $k$ places the system in a unique state, whereas we only require the sequence to cause a unique output. If the specification automaton is in reduced form, it can be shown that the two definitions are equivalent.

## 4.2. Monotonicity Properties

In this section we will prove several properties of the logic simulator that follow from the monotonicity of the simulator functions *Nsim* and *Osim*. As shall be seen, monotonicity provides the primary mechanism by which one can guarantee properties of a circuit knowing only its response during simulation.

**Lemma 1** *The functions SimState and SimOut are monotonic.*

*Proof:* We will prove by induction on the length of $\alpha$ that for any $\alpha, \beta \in \mathcal{TI}$ for which $\alpha \leq \beta$, we have

$$SimState(\alpha) \leq SimState(\beta).$$

First, if $\alpha = \epsilon$, then $SimState(\alpha) = X^s$ and this vector is less than or equal to any other state vector. Otherwise, if $\alpha$ has nonzero length and $\alpha \leq \beta$, then $\alpha$ must be of the form $\alpha' \vec{a}$ and $\beta$ must be of the form $\beta' \vec{b}$ where $\alpha' \leq \beta'$ and $\vec{a} \leq \vec{b}$. Assuming, by induction, that $SimState(\alpha') \leq SimState(\beta')$ and given that *Nsim* is monotonic, we obtain

$$SimState(\alpha) = Nsim(SimState(\alpha'), \vec{a}) \leq Nsim(SimState(\beta'), \vec{b}) = SimState(\beta).$$

The monotonicity of *SimOut* follows from the fact that both *Osim* and *SimState* are monotonic, and the fact that a composition of monotonic functions is also monotonic.

□

The monotonicity of *SimState* and *SimOut* shows how the Erase command and three-valued modeling enhances the power of the simulator. If an Erase command followed by a sequence of Apply commands causes the simulator to produce a 0 or 1 on some output or state variable, then this sequence of inputs must also cause the circuit to produce the same output or state regardless of the initial state. As a special case of this lemma, if sequence $\alpha'$ is a suffix of $\alpha$, then $SimOut(\alpha') \leq SimOut(\alpha)$, i.e., the simulation of the shorter sequence yields an output consistent with, but possibly weaker than, the output produced for the longer one.

**Lemma 2** *If there exists a $k \geq 0$ such that $SimOut(\alpha) = SpecOut(q, \alpha)$ for all $\alpha \in \mathcal{BI}_k$ and all $q \in Q$, then the simulation model fulfills the specification.*

*Proof:*

The assumption of the lemma implies that the specification is $k$-definite and that $SimOut(\alpha) \in B^m$ for all $\alpha \in \mathcal{BI}_k$.

For any $\alpha \in \mathcal{BI}_k$ let

$$Q_\alpha = \{SpecState(q', \alpha) | q' \in Q\}$$

and

$$Z_\alpha = \{\vec{z} \in T^s | \mathit{SimState}(\alpha) \leq \vec{z}\}$$

That is $Q_\alpha$ denotes the set of possible states for the specification automaton following input sequence $\alpha$, while $Z_\alpha$ denotes the set of possible simulator states consistent with the state obtained by simulating the sequence $\alpha$ following an Erase command. By the monotonicity of *Osim*, it follows that for any $\vec{z} \in Z_\alpha$

$$\mathit{SimOut}(\alpha) = \mathit{Osim}(\mathit{SimState}(\alpha)) \leq \mathit{Osim}(\vec{z}).$$

Given that $\mathit{SimOut}(\alpha) \in B^m$, it follows that $\mathit{Osim}(\vec{z}) = \mathit{SimOut}(\alpha)$ for all $\vec{z} \in Z_\alpha$.

Define $\mathcal{E}$ as

$$\mathcal{E} = \bigcup_{\alpha \in \mathcal{BI}_k} \{(q, \vec{z}) | q \in Q_\alpha, \vec{z} \in Z_\alpha\}.$$

We must show that $\mathcal{E}$ satisfies the four properties of Definition 3.

First, since *Nspec* is a surjection, it can be shown by induction on $k$ that for every $q \in Q$, there must be some $\alpha \in \mathcal{BI}_k$ and some $q' \in Q$ such that $q = \mathit{SpecState}(q', \alpha)$. Thus, every state $q$ must be in set $Q_\alpha$ for some $\alpha \in \mathcal{BI}_k$. Moreover, the set $Z_\alpha$ cannot be empty, and hence for every $q \in Q$, there must be some $\vec{z}$ such that $q \; \mathcal{E} \; \vec{z}$.

Second, if $q \; \mathcal{E} \; \vec{z}$, we must have $q = \mathit{SpecState}(q', \alpha)$ and $\mathit{SimState}(\alpha) \leq \vec{z}$ for some $q' \in Q$ and some $\alpha \in \mathcal{BI}_k$. From the condition of the lemma it follows that

$$\mathit{Ospec}(q) = \mathit{SpecOut}(q', \alpha) = \mathit{SimOut}(\alpha) = \mathit{Osim}(\vec{z}).$$

Finally, suppose $q \; \mathcal{E} \; \vec{z}$, i.e., for some $\alpha = [\vec{a}_1, \ldots, \vec{a}_k]$ we have $q \in Q_\alpha$ and $\mathit{SimState}(\alpha) \leq \vec{z}$. Consider any $\vec{x} \in B^p$, and let $\gamma = [\vec{a}_1, \ldots, \vec{a}_k, \vec{x}]$ and $\beta = [\vec{a}_2, \ldots, \vec{a}_k, \vec{x}]$. By definition, $\mathit{Nspec}(q, \vec{x}) \in Q_\beta$. Since $\beta \leq \gamma$ ($\beta$ is a suffix of $\gamma$), and both *SimState* and *Nsim* are monotonic

$$\mathit{SimState}(\beta) \leq \mathit{SimState}(\gamma) = \mathit{Nsim}(\mathit{SimState}(\alpha), \vec{x}) \leq \mathit{Nsim}(\vec{z}, \vec{x}).$$

Therefore $\mathit{Nsim}(\vec{z}, \vec{x}) \in Z_\beta$ by the definition of $Z_\beta$ and hence $\mathit{Nspec}(q, \vec{x}) \; \mathcal{E} \; \mathit{Nsim}(\vec{z}, \vec{x})$.

□

This lemma provides the key to proving that a simulator can verify that a simulation model fulfills a $k$-definite specification by simulating it for all input sequences of length $k$.

**Lemma 3** *If a simulation model fulfills its specification then for all $\alpha \in \mathcal{TI}$ and all $\beta \in \mathcal{BI}$ such that $\alpha \leq \beta$:*

$$\mathit{SimOut}(\alpha) \leq \mathit{SpecOut}(q, \beta)$$

*for all $q \in Q$.*

*Proof*: We will prove by induction on the length of $\beta$ that for some $\vec{z} \in T^s$ such that $SpecState(q, \beta) \; \mathcal{E} \; \vec{z}$, we have $SimState(\beta) \leq \vec{z}$. Given this, we can infer by the monotonicity of *SimOut* and *Osim*, and by the second condition of Definition 3 that

$$SimOut(\alpha) \leq SimOut(\beta) = Osim(SimState(\beta)) \leq Osim(\vec{z}) = SpecOut(q, \beta).$$

To prove the induction hypothesis, for $\beta = \epsilon$, we have that $SimState(\beta) = X^s$. Any $\vec{z}$ such that $q \; \mathcal{E} \; \vec{z}$ therefore satisfies the hypothesis. Now suppose that $\beta$ is of the form $\beta = \beta' \, \vec{b}$, that $SpecState(q, \beta') \; \mathcal{E} \; \vec{z}\,'$, and $SimState(\beta') \leq \vec{z}\,'$. By definition

$$SpecState(q, \beta) = Nspec(SpecState(q, \beta'), \vec{b})$$

and therefore by the third condition of Definition 3

$$SpecState(q, \beta) \; \mathcal{E} \; Nsim(\vec{z}\,', \vec{b}).$$

By the monotonicity of *Nsim*

$$SimState(\beta) = Nsim(SimState(\beta'), \vec{b}) \leq Nsim(\vec{z}\,', \vec{b}).$$

Hence, if we let $\vec{z} = Nsim(\vec{z}\,', \vec{b})$, the induction hypothesis holds.

□

This lemma implies that if the specification has states $q_1$ and $q_2$ for which $SpecOut(q_1, \alpha) \neq SpecOut(q_2, \alpha)$, for some sequence $\alpha$, then some element of $SimOut(\alpha)$ must equal $X$ even for a correctly designed circuit. This property is exploited in our design of an impostor circuit for an indefinite system.

## 4.3. Expressive Power

We are now ready to prove a main result of this paper, characterizing the capabilities and limitations of black-box simulation.

**Theorem 1** *There exists an effective, nontrivial, black-box simulation experiment for a specification if and only if it is definite.*

*Proof*: First, suppose the specification is $k$-definite for some value $k$. Consider the simulation experiment consisting of executing the sequence of commands required to compute $SimOut(\alpha)$ for each $\alpha \in \mathcal{BI}_k$, and accepting the circuit if $SimOut(\alpha) = SpecOut(q, \alpha)$ for all $\alpha$ and any choice of $q \in Q$ (in a $k$-definite specification, the choice of initial state makes no difference.) Lemma 2 shows that this experiment is effective.

Furthermore, the circuit illustrated in Figure 4, consisting of a $p$-bit wide, $k$-bit long shift register to store the most recent $k$ inputs plus logic to compute the circuit outputs can pass

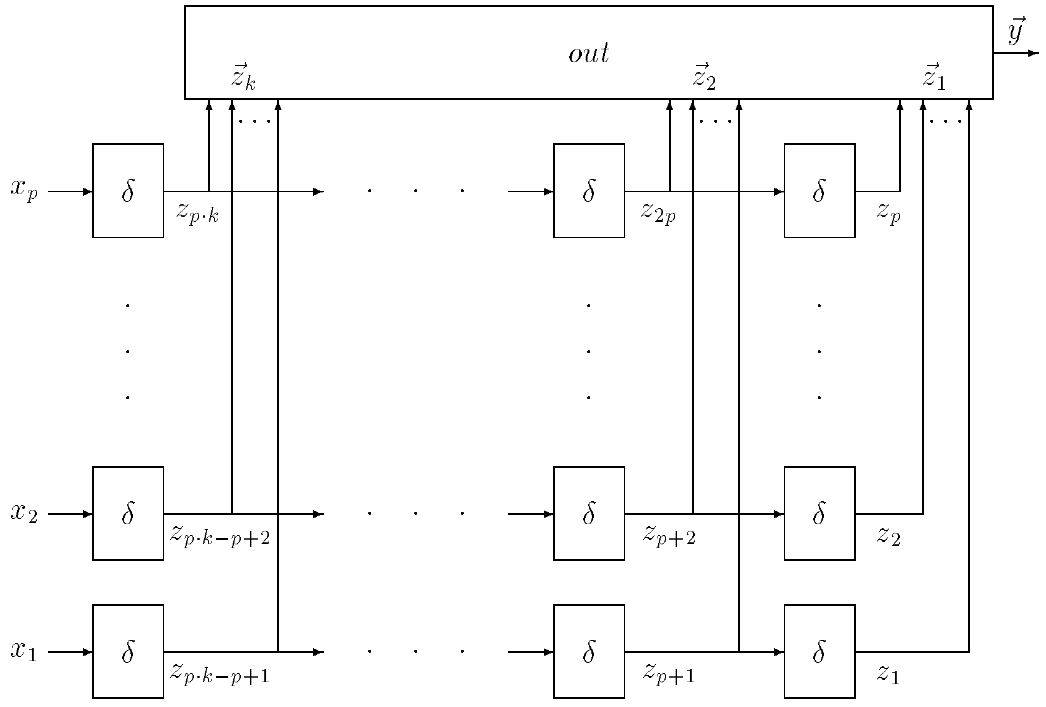

Figure 4: Universal Implementation of a $k$-Definite System

this experiment. A similar structure was proposed by Kleene [15] to implement an arbitrary definite system. More precisely, let $s = p \cdot k$ and define *Nsim* as

$$Nsim_i(\vec{z}, \vec{x}) = \begin{cases} x_{i-p\cdot(k-1)}, & p\cdot(k-1) < i \leq p\cdot k \\ z_{i+p}, & 1 \leq i \leq p\cdot(k-1) \end{cases} \tag{1}$$

Partition $\vec{z}$ into a sequence of vectors $[\vec{z}_1, \ldots, \vec{z}_k]$ where $\vec{z}_i = \langle z_{(i-1)\cdot p+1}, \ldots, z_{i\cdot p}\rangle$ and define *Osim* as

$$Osim(\vec{z}) = \begin{cases} SpecOut(q, [\vec{z}_1, \ldots, \vec{z}_k]), & \vec{z} \in B^s \\ X^m, & \text{otherwise} \end{cases}$$

for any choice of $q \in Q$. It can be seen that for any sequence $\alpha = [\vec{a}_1, \ldots, \vec{a}_k] \in \mathcal{BI}_k$, we have $SimState_{(i-1)\cdot p+j}(\alpha) = a_{i,j}$, the $j$th element of vector $\vec{a}_i$, and hence $SimOut(\alpha) = SpecOut(q, \alpha)$. Therefore, the simulation experiment is nontrivial.

Next, suppose the specification is not $k$-definite for any value of $k$. We use an adversary argument to show that no nontrivial, black-box experiment for this specification can also be effective. Assume that there is some nontrivial simulation experiment in which fewer than $k$ APPLY commands occur without an intervening ERASE command, for some value $k$. For a nontrivial experiment, there must be some simulation model $\mathcal{S} = \langle s, Nsim, Osim\rangle$ that produces an acceptable result for the experiment. We will construct a simulation model $\mathcal{S}' = \langle s', Nsim', Osim'\rangle$ that does not fulfill the specification. This model will have the property that if $SimOut'$ is defined in a manner analogous to the definition of $SimOut$, then $SimOut'(\alpha) = SimOut(\alpha)$ for any sequence $\alpha \in \mathcal{TI}_l$, for which $l < k$. The simulation experiment cannot possibly distinguish $\mathcal{S}$ from $\mathcal{S}'$, and hence it is not effective. This argument holds for any value of $k$, showing that there is no finite upper bound on the length of a

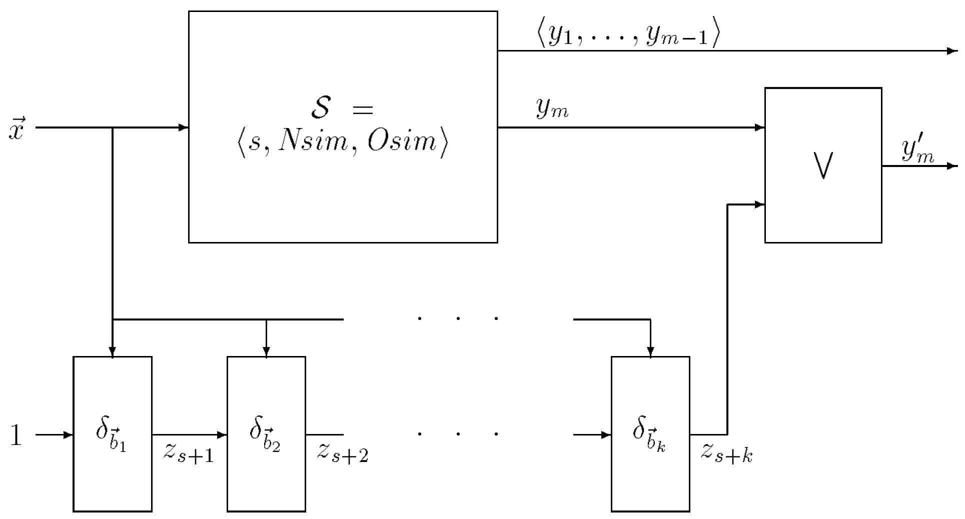

Figure 5: Impostor Simulation Model

simulation experiment that can distinguish a correct circuit from a defective one when the specified system is indefinite.

Simulation model $\mathcal{S}'$ is constructed as illustrated in Figure 5 by taking the circuit from simulation model $\mathcal{S}$ and adding extra logic to implement a "booby trap", i.e., logic that will not affect the output value until a specific input sequence of length $k$ occurs. Designing such a booby trap is no easy task, because any state variables used by the trap will be set to $X$ whenever the user gives an ERASE command. For an improper design this could cause $\mathcal{S}'$ to produce an $X$ on its output under conditions when $\mathcal{S}$ would not. Similarly, the user might attempt to expose any traps by presenting inputs with some elements equal to $X$.

For the design of $\mathcal{S}'$, let $\beta = [\vec{b}_1, \ldots, \vec{b}_k] \in \mathcal{BI}_k$ be some sequence such that the specification has states $q_1, q_2 \in Q$ for which $SpecOut(q_1, \beta) \neq SpecOut(q_2, \beta)$. Such a sequence must exist or else the specification would be $k$-definite. Assume for simplicity, that output $m$ differs for these two cases, relabeling the outputs if required. The trap consists of a shift register, where each shift element sets its output to its input value when the circuit input matches the corresponding element of $\beta$ and clears the output to 0 otherwise. Consequently, input sequence $\beta$ will cause a 1 to propagate through the shift register, forcing output $m$ to 1 when it reaches the end. Any input sequence $\alpha \not\leq \beta$ of length less than or equal to $k$ will cause the shift register to produce 0, leaving the circuit output unchanged. The behavior for input sequences $\alpha < \beta$ will depend on the initial state, and hence under such conditions $SimOut'_m(\alpha) = X$. However, it can be shown using Lemma 3 that under these conditions $SimOut_m(\alpha) = X$ as well.

The detailed design of simulation model $\mathcal{S}'$ is somewhat tedious and hence is described in the appendix. It is also shown that for any $l \leq k$, any $\alpha \in \mathcal{TI}_l$, and any $i$ such that $1 \leq i \leq m$:

$$SimOut'_i(\alpha) = \begin{cases} 1, & \text{if } i = m \text{ and } \alpha = \beta \\ SimOut_i(\alpha), & \text{otherwise} \end{cases}$$

Hence $\mathcal{S}'$ cannot be distinguished from $\mathcal{S}$ for any input sequence of length less than $k$. On the other hand, $\mathit{SpecOut}_m(q_1, \beta) \neq \mathit{SpecOut}_m(q_2, \beta)$, but $\mathit{SimOut}'_m(\beta) = 1$. Therefore, we must have either $\mathit{SimOut}'_m(\beta) \not\sqsubseteq \mathit{SpecOut}_m(q_1, \beta)$ or $\mathit{SimOut}'_m(\beta) \not\sqsubseteq \mathit{SpecOut}_m(q_2, \beta)$ and hence by Lemma 3, $\mathcal{S}'$ cannot fulfill the specification.

$\square$

## 5. State Transition Simulation

To verify circuits implementing a more general class than definite systems, the results of the previous section imply that a capability beyond black-box simulation is required. At the opposite extreme, if the user were to completely specify the relation $\mathcal{E}$, we could check that it satisfies the conditions of Definition 3 by exhaustively simulating all states and transitions. This approach would work for any class of systems and circuits. In practice, however, it cannot be applied to circuits of significant size, because the complexity of completely specifying and checking the relation $\mathcal{E}$ would be overwhelming. Exhaustive simulation, however, provides a basis for developing other simulation methods that overcome the deficiencies of black-box simulation. It shows that any circuit can be verified if we introduce sufficient detail about the circuit structure into the verification method.

### 5.1. The Assertion Method

We would prefer to introduce as little information as possible about the circuit structure into the verification. Toward this goal we will develop a notation similar to the Floyd-Hoare assertion method of program verification, along with an associated simulation methodology for testing assertions.

**Definition 4** *A* circuit assertion *is a set of formulas:*

*Initial:* *a precondition on the state variables,*

*Action:* *a condition on the input variables.*

*Result:* *a postcondition on the state and output variables.*

Each formula is a predicate of the form $L_1 \wedge L_2 \wedge \cdots \wedge L_k$, where each $L_i$ is a *literal* of the form $v = 1$ or $v = 0$, for some circuit variable $v$. Furthermore, no literal may occur in a formula more than once. A circuit *assertion* is written as an equation of the form *Initial* $\{$ *Action* $\}$ *Result*. A simulation model *satisfies* this assertion if for any initial state $\vec{z} \in B^s$ satisfying *Initial* and any input $\vec{x} \in B^p$ satisfying *Action*, the state $\mathit{Nsim}(\vec{z}, \vec{x})$ and the output $\mathit{Osim}(\mathit{Nsim}(\vec{z}, \vec{x}))$ satisfy *Result*.

Given a set of circuit assertions, verifying a circuit requires two proofs—that any simulation model satisfying the set of assertions must fulfill the specification, and that the model under consideration satisfies these assertions. Proving the adequacy of a set of assertions involves showing that they cover every transition in the specification automaton. At the present stage of this research, the set of assertions and a proof of their adequacy must be generated manually. Although this places additional burden on the user, experience has shown that far less manual effort is required than with structural verifiers.

## 5.2. Testing Assertions by Simulation

Once a set of assertions has been devised, a simulator can verify that a particular model satisfies them. The restricted form of the assertion formulas guarantees that, if a vector $\vec{x} \in T^n$ satisfies a formula, then any vector $\vec{x}' \in T^n$ such that $\vec{x} \leq \vec{x}'$ must also satisfy the formula.

For a formula *Initial*, define the vector $\vec{z}_I \in T^s$

$$[\vec{z}_I]_i = \begin{cases} 1, & \text{if the term } z_i = 1 \text{ occurs in } \textit{Initial} \\ 0, & \text{if the term } z_i = 0 \text{ occurs in } \textit{Initial} \\ X, & \text{otherwise.} \end{cases}$$

In a similar fashion, define the vector $\vec{x}_A \in T^p$ according to the terms in *Action*. Observe that $\vec{z}_I$ is the minimum vector satisfying *Initial*, and that $\vec{x}_A$ is the minimum vector satisfying *Action*. These 2 vectors are equivalent to "cube" representations of the 2 formulas [22].

**Theorem 2** *For an assertion Initial { Action } Result define corresponding vectors $\vec{z}_I$, $\vec{x}_A$ according to the formulas Initial and Action. If the vectors $Nsim(\vec{z}_I, \vec{x}_A)$ and $Osim(Nsim(\vec{z}_I, \vec{x}_A))$ satisfy Result, then the simulation model satisfies the assertion.*

*Proof*: Let $\vec{z} \in T^s$ be any state satisfying *Initial*, and $\vec{x} \in B^p$ be any input satisfying *Action*. Clearly, $\vec{z}_I \leq \vec{z}$, and $\vec{x}_A \leq \vec{x}$. By the monotonicity of *Nsim* and *Osim*

$$Nsim(\vec{z}_I, \vec{x}_A) \leq Nsim(\vec{z}, \vec{x}),$$

and

$$Osim(Nsim(\vec{z}_I, \vec{x}_A)) \leq Osim(Nsim(\vec{z}, \vec{x})).$$

and therefore $Nsim(\vec{z}, \vec{x})$ and $Osim(Nsim(\vec{z}, \vec{x}))$ must satisfy *Result*.

□

This theorem indicates a straightforward procedure to test that a simulation model satisfies an assertion. Following an ERASE command, use SET commands to set all state variables occuring in the formula *Initial* to the appropriate values. Then give the command APPLY($\vec{x}_A$) to simulate the prescribed action. Finally, use OBSERVE and OUTPUT commands to check that the value of each state or output variable occuring in *Result* equals its appropriate value.

## 6. Performance Considerations

Up to this point, we have considered only whether verifying a circuit by simulation is even possible. The resulting verification methods are not at all efficient. For example, brute force application of black-box simulation to verify a $k$-definite system with $m$ inputs requires simulating $2^{km}$ patterns. Clearly, this is practical only for small values of $k$ and $m$. In general, the circuit verification problem is NP-hard as measured in the size of the circuit and the specification. However, several techniques reduce the complexity to manageable levels for a large class of circuits.

### 6.1. Input Weakening

The logic value $X$ can be used to indicate a "don't care" (or more properly "shouldn't care") condition when the circuit behavior being tested should not depend on that particular input. This allows us to simulate the effects of a number of Boolean sequences with a single ternary sequence, leading at times to a dramatic reduction in the simulation complexity. This technique is called "input weakening", because it involves reducing the information content of the simulation sequences. Monotonicity guarantees that if the resulting response on some output is 0 or 1, then all stronger sequences would give the same response.

For example, consider a $k$-bit long, 1-bit wide shift register. Brute force, black-box simulation requires simulating $2^k$ patterns of the form $[a_1, a_2, \ldots, a_k]$, each time checking that the final output equals $a_1$. Since the output of the shift register should depend only on the first value in the sequence, we can set the input to $X$ for the remainder of the simulation. This reduces the number of simulation sequences to two: $[1, X, \ldots, X]$ and $[0, X, \ldots, X]$, without compromising the rigor of the simulation. Generalizing this to a shift register of width $m$, a total of $2m$ sequences, each of length $k$, suffices, consisting of a pair to test each bit of the data word. Compared to the *ad hoc* methods most designers use to validate shift registers (e.g., simulate a randomly chosen input sequence), the proposed method provides better results at a comparable cost.

To develop this idea formally, we define a covering set as a set of ternary sequences that include all possible Boolean sequences of a given length. That is, a set $A \subseteq \mathcal{TI}$ is a *covering set* for $\mathcal{BI}_k$ if to every $\beta \in \mathcal{BI}_k$ there corresponds some $\alpha \in A$ such that $\alpha \leq \beta$.

**Theorem 3** *For a covering set $A$ of $\mathcal{BI}_k$, if $SimOut(\alpha) = SpecOut(q, \beta)$ for all $\alpha \in A$, all $\beta \in \mathcal{BI}_k$ such that $\alpha \leq \beta$, and all $q \in Q$, then the simulation model fulfills the specification.*

*Proof*: By the monotonicity of $SimOut$, if $\alpha \leq \beta$, then $SimOut(\alpha) \leq SimOut(\beta)$. The assumption that $SimOut(\alpha) = SpecOut(q, \beta)$ implies that $SimOut(\alpha) \in B^m$, and hence $SimOut(\beta) = SpecOut(q, \beta)$. Thus, the conditions required by Lemma 2 hold.

□

Input weakening can also be applied in transition simulation. In fact the simulation sequences arising from the assertion method already utilize this technique. If the formulas *Initial* and *Action* place no conditions on an input or state variable, then it is set to $X$ in the simulation.

## 6.2. Symbolic Simulation

At times we cannot avoid the complexity caused by the large number of possible input combinations that might be applied to a circuit, all of which might be relevant to the values of the outputs. For these cases, we propose *symbolic simulation* to reduce the number of patterns simulated. A symbolic simulator [5] resembles a conventional logic simulator, except that the input sequences can contain Boolean variables in addition to the constants 1 and 0. During simulation the values of the circuit state and output are Boolean functions of the variables occurring in the input sequence. A symbolic simulator represents and manipulates these functions explicitly. The worst-case behavior of such a program gives no better performance than exhaustive simulation by a conventional simulator. However, good Boolean manipulation algorithms often lead to far better results. To implement the verification methodologies described in this paper, a symbolic simulator must be able to manipulate functions over the three-valued domain $\{0, 1, X\}$. The symbolic simulator MOSSYM [5] solves this problem by representing every circuit variable by a pair of Boolean functions, generalizing the encoding of three possible values by two bits.

A symbolic simulator can verify an $m$-input, $k$-definite system by simulating a single sequence of length $k$, with each input pattern consisting of $m$ Boolean variables to represent all possible input values. The resulting output functions will be symbolic representations of the circuit outputs for every possible input sequence of length $k$. These can then be tested for equivalence with functions generated from the system specification. As an example of verification by symbolic simulation, the above-mentioned shift register would be verified by simulating the sequence of variables $[a_1, a_2, \ldots, a_k]$ and testing the final output for equivalence with the function $a_1$. Efficient symbolic manipulation will exploit the fact that the variables shifting through the register do not interact. Hence a symbolic simulator can automatically take advantage of the same properties that allow input weakening. Symbolic simulation can also handle cases for which input weakening does not apply. For example, MOSSYM was able to verify a 16-bit nMOS adder using less than 10 minutes of CPU time on a Digital Equipment Corporation VAX-11/780. In contrast, its more traditional counterpart MOSSIM II [4] would require an estimated 648 years using exhaustive black-box simulation.

The capabilities of symbolic simulators can also be exploited when verifying indefinite systems. Rather than testing a large number of assertions with formulas containing terms of the form $v = 0$ or $v = 1$, the program would test a smaller set of assertions where the formulas contain terms of the form $v = a$ where $a$ is a universally-quantified Boolean variable.

Although a symbolic simulator gives the user a far more abstract view of circuit behavior, it has no fundamental power beyond that of an ordinary logic simulator. Any information that symbolic simulation provides could also be obtained by exhaustively simulating the set of patterns generated by enumerating all combinations of 0 and 1 for the Boolean variables.

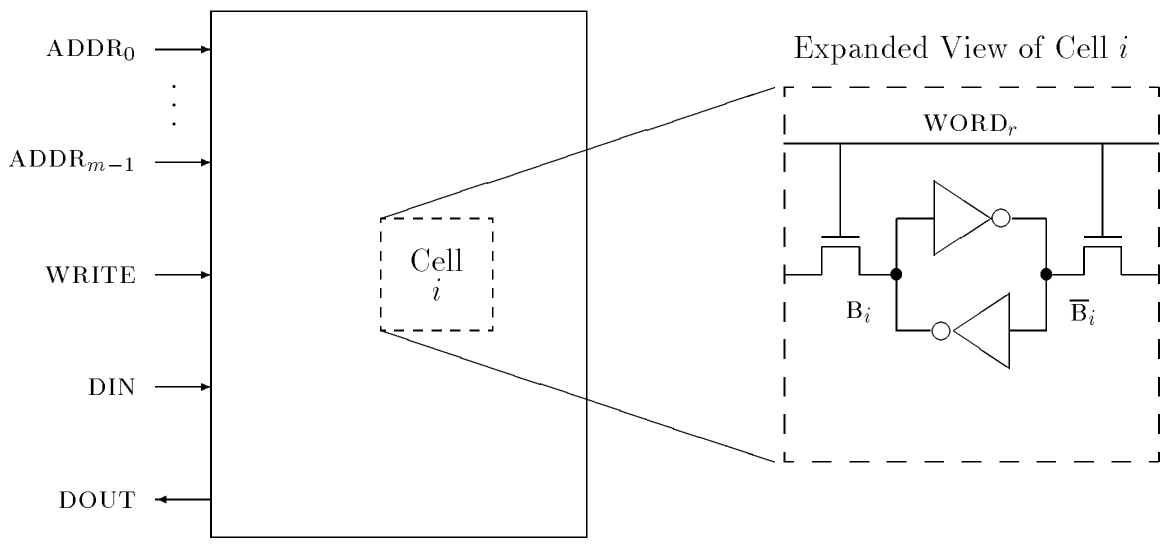

Figure 6: Static RAM Circuit

Hence, the results of this paper provide a foundation for verifying circuits by symbolic, as well as by traditional, simulation.

## 7. Memory Verification Example

To demonstrate the methodology on a more significant task for hardware verification, consider the static random-access memory (RAM) illustrated in Figure 6. The assertions required to verify this circuit will be presented in a series of steps, each introducing new notation and discussing the reasoning behind it. Despite these extensions to the notation, the underlying principle remains that of state transition simulation. This verification methodology has successfully been applied to a 4096-bit memory [6].

This circuit holds $n = 2^m$ bits, where each memory cell $i$, such that $0 \leq i < n$, consists of a feedback path containing electrical nodes $\text{B}_i$ and $\overline{\text{B}}_i$ along with a pair of access transistors [11]. As a shorthand, the predicate *Stored*$(i, v)$ expresses the fact that value $v \in \{0, 1\}$ is stored in memory cell $i$:

$$\mathit{Stored}(i, v) \quad \equiv \quad \text{B}_i = v \ \wedge \ \overline{\text{B}}_i = \neg v.$$

The input lines $\text{ADDR}_j$, for $0 \leq j < m$, select a particular memory cell. When WRITE $= 1$, the value of DIN is written into the selected memory cell. As shorthand, this operation is expressed by a predicate

$$\mathit{Write}(i, v) \quad \equiv \quad \text{WRITE} = 1 \ \wedge \ \text{DIN} = v \ \wedge \ \forall (0 \leq k < m)[\text{ADDR}_k = i_k]$$

where $i_k$ indicates the $k$th bit in the binary representation of $i$. When WRITE $= 0$, the value stored in the selected memory cell is produced as output on DOUT. This operation is expressed by the predicate

$$\mathit{Read}(i) \quad \equiv \quad \text{WRITE} = 0 \ \wedge \ \forall(0 \leq k < m)[\text{ADDR}_k = i_k].$$

Although few additional details of the circuit design are needed for verification, correct circuit operation depends on the fact that the control lines $\text{WORD}_r$ return to 0 between memory operations, for $0 \leq r < \sqrt{n}$.[1] Without this property, the access transistors for more than one cell in a column could be turned on, causing undesirable interactions. This fact is formulated as a *system invariant*

$$\mathit{Inv} \quad \equiv \quad \forall(0 \leq r < \sqrt{n})[\text{WORD}_r = 0].$$

The invariance of this condition is expressed by a single assertion:

$$\mathit{true} \left\{ \ \mathit{true} \ \right\} \mathit{Inv}$$

That is, following any memory operation, the word lines will return to a quiescent condition. Once the assertion has been established, the invariant *Inv* can be assumed as a precondition in all other assertions. Most circuits require some form of system invariant expressing conditions about the control logic that can be assumed true at the beginning of every input cycle. Devising the invariant requires a combination of analysis and experimentation. An insufficient system invariant will become immediately apparent during subsequent simulations, because output or state variables that should have Boolean values will equal $X$.

The remaining assertions simply express the operation of a memory. First, for all $v \in \{0, 1\}$ and for all $i$ such that $0 \leq i < n$, an assertion states that writing $v$ into location $i$ must cause $v$ to be stored in cell $i$:

$$\mathit{Inv} \left\{ \ \mathit{Write}(i, v) \ \right\} \mathit{Stored}(i, v).$$

Second, for all $v \in \{0, 1\}$ and for all $i$ and $j$ such that $0 \leq i, j < n$ and $i \neq j$, an assertion states that writing into location $j$ does not affect the value in cell $i$:

$$\mathit{Inv} \ \wedge \ \mathit{Stored}(i, v) \left\{ \ \mathit{Write}(j, X) \ \right\} \mathit{Stored}(i, v). \tag{2}$$

Third, for all $v \in \{0, 1\}$ and for all $i$ such that $0 \leq i < n$, an assertion states that reading location $i$ causes its value to appear on the output:

$$\mathit{Inv} \ \wedge \ \mathit{Stored}(i, v) \left\{ \ \mathit{Read}(i) \ \right\} \text{DOUT} = v.$$

Finally, for all $v \in \{0, 1\}$ and for all $i$ such that $0 \leq i < n$, an assertion states that reading a value from any location should have no effect on the value stored in location $i$:

$$\mathit{Inv} \ \wedge \ \mathit{Stored}(i, v) \left\{ \ \text{WRITE} = 0 \ \right\} \mathit{Stored}(i, v). \tag{3}$$

[1]A memory circuit is generally configured as a square array of memory cells with $m/2$ of the address bits selecting a row and the remaining selecting a column. Hence there are $\sqrt{n}$ rows.

The above equations represent a total of $2n^2 + 4n + 1$ assertions. The number can be further reduced by exploiting input weakening for the cases covered by Equation 2. That is, for an address $i$ with bit representation $\langle i_0, \ldots, i_{m-1} \rangle$, all addresses $j$ such that $j \neq i$ are covered by vectors of the form $\langle X, \ldots, X, \neg i_k, X, \ldots, X \rangle$ for $0 \leq k < m$. Thus, Equation 2 can be replaced by the following set of assertions for $v \in \{0, 1\}$, $0 \leq i < n$, and $0 \leq k < m$:

$$Inv \wedge Stored(i, v) \left\{ \text{WRITE} = 1 \wedge \text{ADDR}_k = \neg i_k \right\} Stored(i, v). \tag{4}$$

This reduces the total number of assertions to $2n \log n + 4n + 1$. In practice, many memory circuits would yield false negative responses for some of the assertions of Equations 3 and 4. The simulation of an assertion that causes the word line of a memory cell to be set to $X$ would most likely corrupt the value stored in the cell. With more care, however, the set of assertions can be refined to avoid this problem while maintaining the $O(n \log n)$ bound on the total number of patterns to be simulated [6]. These refined patterns first verify that a memory cell is not affected by an operation on a cell in another row, and then that it is not affected by an operation on a cell in a different column of the same row.

Considering that even a minimal validation of a memory circuit requires simulating $\Omega(n)$ patterns (e.g., read and write every memory location), simulating $O(n \log n)$ patterns seems a very reasonable price to pay for rigorous verification. The efficiency of this verification results from an extreme form of input (and state variable) weakening. The verification isolates each memory location, proving that it can be written and read properly, and that operations on other memory locations do not corrupt its stored value. During each test, those memory locations not under consideration are set to $X$. Should the circuit contain an undesirable pattern sensitivity, at least one of the tests will fail with an output or state variable equal to $X$ that should equal 0 or 1.

Devising a simulation sequence to verify this memory requires paying to the details of the circuit design, especially when trying to minimize the number of assertions. In contrast to black-box simulation, the resulting simulation sequence is highly circuit dependent. Compare this effort, however, to that required by other verifiers. For structural verification the user would be required to specify the operation of all aspects of the circuit including the address decoders, bit lines, sense amps, and control logic. These specifications would require a circuit model that can express such effects as bidirectional transistor behavior, ratioed circuits, and precharged logic. By comparison, behavioral verification seems quite straightforward.

## 8. Discussion

This paper has outlined a method for applying three-valued logic simulation to the task of hardware verification. Complex circuits can be rigorously verified given only information about the desired input-output behavior, and possibly some information about the circuit state variables.

Multi-valued modeling provides the fundamental mechanism by which a circuit can be verified knowing little about its internal structure. The requirements placed on the simulator to support verification are fairly mild. Most contemporary logic simulators provide a value $X$ to avoid the need to find an initial Boolean state of the circuit that does not cause oscillations [14]. Although an explicit ERASE command may not be provided, it can easily be implemented, or the same result can be obtained by simply restarting the program. The monotonicity requirement simply expresses the desirable property that in the presence of $X$ values, the simulator should not set an output or state variable to 0 or 1, when this would not have occurred had some of the $X$'s been 0 or 1 instead. Any reasonable implementation satisfies this.

The resulting simulation sequences, however, differ greatly from those commonly used by circuit designers during informal validation. In particular, the state of the circuit is frequently set to all $X$'s so that any accidental sequential dependencies will be detected. During most sequences, only a small number of state or input variables are set to Boolean values, and attention is focused on the effect these values have on the output or new circuit state. Any accidental dependencies on other state or input variables will manifest themselves as $X$'s on output or state variables that are expected to have Boolean values. Most logic simulators have not been designed for this style of simulation. Many use pessimistic methods of computing the effect of $X$ values, causing them to produce $X$'s where it can be shown that the true results should be Boolean values. Such a simulator provides too dull a tool for formal circuit verification, giving many false negative responses. With greater care, however, simulators can be designed to provide more accurate and efficient modeling of $X$'s.

This methodology demonstrates several worthwhile simulation practices that could be applied even when formal verification is not sought. For example, typical simulation runs consist of many pattern sequences where the behavior of the circuit should not depend on the relative ordering of these sequences. Preceding each pattern sequence by an ERASE command would help uncover any accidental pattern sensitivities. As the memory verification example showed, a simulator can uncover more potential errors if the user can focus on small regions of the circuit at a time, setting to $X$ those input and state variables that should not affect the behavior in this region. A common practice followed by circuit designers today is to simulate an enormous number of patterns, possibly consuming weeks of CPU time, hoping that brute force will uncover any error. By following a more disciplined methodology, shorter simulation runs could be devised that yield more reliable results.

## Appendix. Design of Impostor Simulation Model

This appendix documents a simulation model that cannot be distinguished from a model that fulfills an indefinite system specification for any sequence of length less than $k$, as required in the proof of Theorem 1. We assume in this design that $\beta$ is some sequence $[\vec{b}_1, \ldots, \vec{b}_k] \in \mathcal{BI}_k$ such that $\mathit{SpecOut}_m(q_1, \beta) \neq \mathit{SpecOut}_m(q_2, \beta)$ for two states $q_1, q_2 \in Q$. Let $s' = s + k$. Referring to Figure 5 the shift register elements in the booby trap each have two sets of inputs: a single data input $t$ and a set of control inputs $\vec{a}$. For vector $\vec{b} \in T^p$, define the function of a shift register element $\delta_{\vec{b}}: T \times T^p \rightarrow T$ as

$$\delta_{\vec{b}}(t, \vec{a}) = \begin{cases} t, & \vec{a} = \vec{b} \\ 0, & \vec{a} \not\leq \vec{b} \text{ or } t = 0 \\ X, & \vec{a} < \vec{b} \text{ and } t \neq 0 \end{cases} \tag{5}$$

That is, the input data is shifted to the output when the control inputs match those given by vector $\vec{b}$. The output is cleared to 0 when at least one control input differs from the corresponding element of $\vec{b}$ but does not equal $X$. To satisfy monotonicity, we adopt the convention that whenever $\vec{a} < \vec{b}$ the output equals 0 only if the data input equals 0, i.e., it does not matter whether the input is shifted or the output is cleared, and equals $X$ otherwise.

The next state function for $\mathcal{S}'$ is defined as

$$Nsim'_i(\vec{z}, \vec{x}) = \begin{cases} Nsim_i(\langle z_1, \ldots z_s \rangle, \vec{x}), & i \leq s \\ \delta_{\vec{b}_1}(1, \vec{x}), & i = s+1 \\ \delta_{\vec{b}_{i-s}}(z_{i-1}, \vec{x}) & s+2 \leq i \leq s+k \end{cases} \tag{6}$$

The output function is defined as

$$Osim'_i(\vec{z}, \vec{x}) = \begin{cases} Osim_i(\langle z_1, \ldots, z_s \rangle, \vec{x}), & i < m \\ z_{s+k} \vee Osim_m(\langle z_1, \ldots, z_s \rangle, \vec{x}), & i = m \end{cases} \tag{7}$$

where $\vee$ is defined according to the leftmost case in Table 1.

To prove that $\mathcal{S}'$ behaves as claimed, we require the following property about the value produced by the shift register for a given input sequence.

**Lemma 4** *For any $l \leq k$ and any $\alpha \in \mathcal{TI}_l$,*

$$SimState'_{s+k}(\alpha) = \begin{cases} 1, & \alpha = \beta \\ 0, & \alpha \not\leq \beta \\ X, & \alpha < \beta \end{cases}$$

*Proof*: We will show by induction on $i$ that for any $l, i$ such that $1 \leq l \leq i \leq k$, if we consider any sequence $\alpha \in \mathcal{TI}_l$ then

$$SimState'_{s+i}(\alpha) = \begin{cases} 1, & \alpha = [\vec{b}_1, \ldots \vec{b}_i] \\ 0, & \alpha \not\leq [\vec{b}_1, \ldots \vec{b}_i] \\ X, & \alpha < [\vec{b}_1, \ldots \vec{b}_i] \end{cases} \tag{8}$$

The statement of the lemma then holds by letting $i = k$.

First, suppose $i = 1$, in which case either $\alpha = \epsilon$ whereby $\alpha < [\vec{b}_1]$ and $SimState'_{s+1}(\alpha) = X$, or $\alpha = [\vec{a}_1]$ for some $\vec{a}_1 \in T^p$, whereby $SimState'_{s+1}(\alpha) = \delta_{\vec{b}_1}(1, \vec{a}_1)$. Comparing Equation 5 with $t = 1$ to Equation 8 we see that the desired condition holds.

Now assume that Equation 8 holds for some value $i$. Let $\alpha = \alpha' \, \vec{a}_{l+1}$ be a sequence where $\alpha' \in \mathcal{TI}_l$ and $l \leq i$. Consider the ways $\alpha$ can relate to the sequence $[\vec{b}_1, \ldots, \vec{b}_{i+1}]$ in Equation 8.

Equality can hold only if $l = i$ and both $\alpha' = [\vec{b}_1, \ldots, \vec{b}_i]$ and $\vec{a}_{l+1} = \vec{b}_{i+1}$. Combining Equations 5, 6, and 8 for this case we get

$$\mathit{SimState}'_{s+i+1}(\alpha) = \delta_{\vec{b}_{i+1}}(\mathit{SimState}'_{s+i}(\alpha'), \vec{a}_{l+1}) = \delta_{\vec{b}_{i+1}}(1, \vec{b}_{i+1}) = 1.$$

Incomparability, i.e., $\alpha \not\lesssim [\vec{b}_1, \ldots, \vec{b}_{i+1}]$ can hold only if either $\vec{a}_{l+1} \not\lesssim \vec{b}_{i+1}$ or $\alpha' \not\lesssim [\vec{b}, \ldots, \vec{b}_i]$. In the first case we have

$$\mathit{SimState}'_{s+i+1}(\alpha) = \delta_{\vec{b}_{i+1}}(\mathit{SimState}'_{s+i}(\alpha'), \vec{a}_{l+1}) = \delta_{\vec{b}_{i+1}}(t, \vec{a}_{l+1}) = 0.$$

In the second case we have

$$\mathit{SimState}'_{s+i+1}(\alpha) = \delta_{\vec{b}_{i+1}}(\mathit{SimState}'_{s+i}(\alpha'), \vec{a}_{l+1}) = \delta_{\vec{b}_{i+1}}(0, \vec{a}_{l+1}) = 0.$$

The sequences may be ordered $\alpha < [\vec{b}_1, \ldots, \vec{b}_{i+1}]$ only if either $\alpha' < [\vec{b}, \ldots, \vec{b}_i]$ and $\vec{a}_{l+1} \leq \vec{b}_{i+1}$, in which case

$$\mathit{SimState}'_{s+i+1}(\alpha) = \delta_{\vec{b}_{i+1}}(\mathit{SimState}'_{s+i}(\alpha'), \vec{a}_{l+1}) = \delta_{\vec{b}_{i+1}}(X, \vec{a}_{l+1}) = X,$$

or $\alpha' = [\vec{b}, \ldots, \vec{b}_i]$ and $\vec{a}_{l+1} < \vec{b}_{i+1}$ in which case

$$\mathit{SimState}'_{s+i+1}(\alpha) = \delta_{\vec{b}_{i+1}}(\mathit{SimState}'_{s+i}(\alpha'), \vec{a}_{l+1}) = \delta_{\vec{b}_{i+1}}(1, \vec{a}_{l+1}) = X.$$

Finally, the sequences cannot be ordered $\alpha > [\vec{b}_1, \ldots, \vec{b}_{i+1}]$, because $l \leq i$, and $[\vec{b}_1, \ldots, \vec{b}_{i+1}] \in \mathcal{BI}$

$\square$

**Lemma 5** *For any $l \leq k$, any $\alpha \in \mathcal{TI}_l$, and any $i$ such that $1 \leq i \leq m$:*

$$\mathit{SimOut}'_i(\alpha) = \begin{cases} 1, & \text{if } i = m \text{ and } \alpha = \beta \\ \mathit{SimOut}_i(\alpha), & \text{otherwise.} \end{cases}$$

*Proof*: For all cases except where $i = m$ and $\alpha < \beta$, this result follows from the definition of $\mathit{Osim}'$ and from Lemma 4. For $\alpha < \beta$, Lemma 4 tells us that the shift register output will equal $X$. However, given that $\mathcal{S}$ fulfills the specification, Lemma 3 shows that both $\mathit{SimOut}_m(\alpha) \leq \mathit{SpecOut}_m(q_1, \beta)$ and $\mathit{SimOut}_m(\alpha) \leq \mathit{SpecOut}_m(q_2, \beta)$. Since these two values are unequal, we must have

$$\mathit{SimOut}_m(\alpha) = \mathit{SimOut}'_m(\alpha) = X.$$

$\square$

## References

[1] Barrow, H. G. Proving the correctness of digital hardware designs. *VLSI Design V*, 7 (July 1984), 64–77.

[2] Barrow, H. G. VERIFY: a program for proving correctness of digital hardware designs. *Artificial Intelligence 24* (1984), 437–491.

[3] Browne, M. C., Clarke, E. M., Dill, D. L., and Mishra, B. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers C-35*, 12 (Dec. 1986), 1035–1044.

[4] Bryant, R. E. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers C-33*, 2 (Feb. 1984), 160–177.

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

[6] R. E. Bryant, "Verifying a Static RAM Design by Logic Simulation," *Fifth MIT Conference on Advanced Research in VLSI*, 1988, 335–349.

[7] Brzozowski, J. A., and Yoeli, M. On a ternary model of gate networks. *IEEE Transactions on Computers C-28*, 3 (March 1979), 178–183.

[8] Dill, D. L., and Clarke, E. M. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings 133*, Pt. E, 5 (Sept. 1986), 276–282.

[9] Floyd, R. W. "Assigning meanings to programs," *Proc. Symp. in Applied Mathematics, 19—Mathematical Aspects of Computer Science*, Schwartz, J. T., Ed. AMS, 1967, 19–32.

[10] German, S. M., and Wang, Y. Formal verification of parameterized hardware designs. *Int. Conf. on Computer Design*, IEEE, 1985, 549–552.

[11] Glasser, L. A., and Dobberpuhl, D. W. *The Design and Analysis of VLSI Circuits*, Addison-Wesley, Reading, MA, 1985.

[12] Hartmanis, J, and Stearns, R. E. *Algebraic structure theory of sequential machines*, Prentice-Hall, Englewood Cliffs, NJ, 1966.

[13] Hoare, C. A. R. An axiomatic basis for computer programming. *Comm. ACM 12* (1969), 576–580.

[14] Jephson, J. S., McQuarrie, R. P., and Vogelsberg, R. E. A three-level design verification system. *IBM Systems Journal 8*, 3 (1969), 178–188.

[15] Kleene, S. C. Representation of events in nerve nets and finite automata. *Automata Studies*, Shannon, C. E., and McCarthy, J., Ed. Princeton University Press, Princeton, NJ, 1956, 3–41.

[16] Kohavi, Z. *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1970.

[17] Lengauer, T., and Näher, S. An analysis of ternary simulation as a tool for race detection. *Integration, the VLSI Journal 4*, 4 (Dec. 1986), 309–330.

[18] Milne, G. J. CIRCAL: a calculus for circuit description. *Integration 1*, 2&3 (Oct. 1983) 121–160.

[19] Milne, G. J. A model for hardware description and verification. *21st Design Automation Conference*, ACM and IEEE, 1984.

[20] Moore, E. F. Gedanken-experiments on sequential machines. *Automata Studies*, Shannon C. E., and McCarthy, J., Ed. Princeton University Press, Princeton, NJ, 1956, 129–153.

[21] Perles M., Rabin M. O., and Shamir E. The theory of definite automata. *IEEE Transactions on Electronic Computers EC-12*, 6 (June, 1963), 233–243.

[22] Roth, J. P. *Computer Logic, Testing, and Verification*, Computer Science Press, Rockville, MD, 1980.

[23] Shostak, R. E. Verification of VLSI designs. *Proceedings of the Third Caltech Conference on VLSI*, Bryant, R., Ed. Computer Science Press, Rockville, MD, 1983, 185–206.

[24] Wagner, T. J. *Hardware Verification.* Ph.D. Thesis, Dept. Comp. Sci., Stanford Univ., 1977.

[25] Weise, D. *Automatic Formal Verification of Synchronous MOS VLSI Designs*. Ph.D. Thesis, Dept. Elec. Eng. and Comp. Sci., Massachusetts Inst. of Tech., 1986.

[26] Weise, D. Verifying MOS circuits, *24th Design Automation Conference*, ACM and IEEE, 1987.

[27] Yoeli, M., and Rinon, S. Application of ternary algebra to the study of static hazards, *J. ACM* 11, 1 (Jan. 1964), 84–97.