Formal Verification of Digital Circuits Using Symbolic Ternary System Models*

Randal E. Bryant
Carl-Johan Seger
Carnegie Mellon University

Abstract

Ternary system modeling involves extending the traditional set of binary values $\{0, 1\}$ with a third value *X* indicating an unknown or indeterminate condition. By making this extension, we can model a wider range of circuit phenomena. We can also efficiently verify sequential circuits in which the effect of a given operation depends on only a subset of the total system state.

This paper presents a formal methodology for verifying synchronous digital circuits using a ternary system model. The desired behavior of the circuit is expressed as assertions in a notation using a combination of Boolean expressions and temporal logic operators. An assertion is verified by translating it into a sequence of patterns and checks for a ternary symbolic simulator. This methodology has been used to verify a number of full scale circuit designs.

1. Introduction

1.1. Ternary Modeling

Most formal models for hardware verification assume that every signal always has a well-defined, discrete value. For example, a *binary* model assumes that each signal must be either 0 or 1. In this paper we present a methodology for formal verification in which a third value *X* is added to the set of possible signal values, indicating an unknown or indeterminate logic value. By shifting to a ternary system model, we gain several advantages.

As a first advantage, this extension makes it possible to model an increased range of circuit phenomena. For example, we can deal with circuits in which nondigital voltages are generated

*This research was supported by the Defense Advanced Research Projects Agency, ARPA Order Number 4976, and by the National Science Foundation, under grant number MIP-8913667.

in the course of normal circuit operation. This occurs frequently when modeling circuits at the switch-level [6], due to (generally transient) short circuits or charge sharing. We can also deal with circuits in which indeterminate behavior occurs due either to timing hazards or to circuit oscillation. In all of these cases, the modeling algorithm expresses this uncertainty by assigning value X to the offending circuit nodes, indicating that the actual digital value cannot be determined [8, 15].

As a second advantage, we can efficiently verify many aspects of digital circuit behavior by representing the circuit with a ternary system model. We do this by *ternary symbolic simulation*, in which a simulation algorithm designed to operate on scalar values 0, 1, and X, is extended to operate on a set of symbolic values. Each symbolic value indicates the value of a signal for many different operating conditions, parameterized in terms of a set of symbolic Boolean variables. Since the value X indicates that a signal could be either 0 or 1 (or a non-digital voltage), we can often represent many different operating conditions by the constant value X, rather than with a more complex symbolic value. For example, we can verify that a particular sequence of actions will yield a 1 (or 0) on some node regardless of the initial state by verifying that this value results when starting from an initial state where all nodes are set to X. This requires far less effort than analyzing the effect of the action on all possible initial binary states.

Simulators that support ternary modeling intentionally err on the side of pessimism for the sake of efficiency. That is, they will sometimes produce a value X even where exhaustive case analysis would indicate that the value should be binary (i.e., 0 or 1). For example, most ternary simulators evaluate logic functions in a ternary algebra created by extending the standard Boolean operators. This algebra does not obey the law of excluded middle, because $X +_t \overline{X}^t = X$, where $+_t$ and $^{-t}$ are ternary extensions of Boolean sum and complement, respectively. On the other hand, symbolic simulation avoids this pessimism, because it can resolve the interdependencies among signal values, and compute $a + \overline{a} = \mathbf{1}$ (the Boolean function that always yields 1). By combining the expressive power of symbolic values with the computational efficiency of ternary values, we can trade off precision for ease of computation.

When creating a ternary system model, we impose the following monotonicity requirement: In the presence of X signals as stimuli, no action of the circuit should yield a binary value, unless the same value would occur if any subset of these stimuli had binary values instead. That is, when given incomplete information about the exact circuit state, we should never produce a signal inconsistent with one that would result if more information were available. This monotonicity condition makes it possible to verify properties of circuit operation in the presence of potential sources of indeterminate behavior by representing this indeterminacy with the value X. It also makes it possible to verify properties under a ternary system model as a means of proving properties under a binary system model.

1.2. Contribution of Paper

In earlier work, we demonstrated the utility of ternary modeling for verifying a variety of circuits [1, 7]. Our methodology is based on ternary simulation, either scalar or symbolic. With a simulator we can verify assertions specifying a postcondition on the circuit state that would result given some precondition on the previous state and some condition imposed on the input values. By restricting the form of the precondition, postcondition, and input condition to specifying that some of the nodes have particular binary values, we can verify such an assertion by a single simulation sequence. That is, we perform a simulation in which any node that is constrained by the precondition or action is set to its specified value, while all other nodes are set to *X*. If the resulting node states satisfy those specified by the postcondition, then the assertion is proved. For some classes of circuits, e.g., random access memories, we can verify correct circuit operation for all possible inputs and initial states by simulating only a polynomial number of scalar patterns. For other circuits, we can overcome the combinatorial complexity of considering many different initial states and input actions via symbolic simulation.

This earlier work demonstrated the viability of circuit verification by symbolic simulation, but it fell short in terms of generality, ease of use, and degree of automation. We did not have a formal notation for specifying the desired circuit properties, nor a method to generate simulation patterns directly from the specification. Instead, we derived symbolic simulation patterns by hand and argued informally that these patterns served to verify the desired properties. Furthermore, it was particularly cumbersome to verify operations requiring multiple state transitions, such as occurs in pipelined systems.

In this paper, we correct this shortcoming by presenting a formal state transition model for a ternary system, a formal syntax for expressing desired properties of the system, and an algorithm to decide whether or not the system obeys the specified property. Our state transition system is quite general, and is compatible with a number of circuit modeling techniques.

Our specifications take the form of *symbolic trajectory formulas* mixing Boolean expressions and the temporal *next-time* operator. The Boolean expressions provide a convenient means of describing many different operating conditions in a compact form. For example, we can express the desired behavior of an ALU on all possible inputs. By allowing only the most elementary of temporal operators, the class of properties we can express is relatively restricted, as compared to other temporal logics [10, 17]. In particular, we can only reason about circuit operations involving a bounded number of state transitions. Nonetheless, we have found that we can readily express many aspects of synchronous digital systems. It is quite adequate for expressing many of the subtleties of system operation, including clocking conventions and pipelining.

Our decision algorithm is based on ternary symbolic simulation. It tests the validity of an assertion of the form $[A \Longrightarrow C]$, where both A and C are trajectory formulas. That is, it determines whether or not every state sequence satisfying A (the "antecedent") must also satisfy C (the "consequent"). It does this by generating a symbolic simulation sequence corresponding to the antecedent, and testing whether the resulting symbolic state sequence satisfies the consequent.

An important property of our algorithm is that it requires a comparatively small amount of simulation and symbolic manipulation to verify an assertion. The restrictions we impose on the formula syntax guarantee that there is a unique weakest symbolic sequence satisfying the antecedent. Furthermore, the symbolic manipulations involve only variables explicitly mentioned in the assertion. Unlike other symbolic circuit verifiers [3], we do not need to introduce extra variables denoting the initial circuit state or possible primary inputs. Finally, the length of the simulation sequence depends only on the depth of nesting of temporal next-time operators in the assertion.

Some insight into the expressive power of our specification language can be gained by reviewing the different classes of properties examined by Pnueli in his original paper on temporal logic [17]. In particular, our notation is adequate for expressing many *invariant* properties of the system. That is, we formulate an assertion where the antecedent expresses a set of guaranteed constraints on the circuit inputs and state over multiple time intervals, while the consequent expresses some additional desired constraints on the state. These desired constraints generally include the state guarantees of the antecedent shifted to a later time. By verifying such an assertion, we prove that the desired properties of the consequent hold over the entire operation of the circuit as long as the inputs are supplied correctly. On the other hand, properties that require reasoning about *eventuality* cannot be expressed with our notation, unless the desired property is guaranteed to hold within a bounded time.

1.3. Overview of Implementation

By modifying the COSMOS symbolic simulator[5], we have been able to implement the algorithm described in this paper and to verify several full scale circuit designs. COSMOS represents a MOS circuit at the switch level as a network of transistor switches connecting charge storage nodes. The simulator preprocesses the circuit to transform it into a Boolean representation [6]. Using this Boolean representation, the switch-level behavior of the circuit can be computed by simply evaluating a series of Boolean operations, much as one would in a gate-level circuit model. The preprocessor is quite general and accurate. The resulting representation is guaranteed to produce the exact same results as more traditional switch-level simulators.

Simulating a circuit symbolically involves evaluating these Boolean operations over the algebra generated by a set of Boolean variables. That is, each element of the algebra corresponds to a Boolean function over the variables. By representing these functions as Ordered Binary Decision Diagrams [4], complex functions can be represented and manipulated efficiently. Of course, all of the verification properties we wish to decide require solving NP-hard problems. Our approach has a worst case time and space requirement that grows exponentially with the size of the formulas expressing the circuit property to be decided. Nonetheless, we have been successful at avoiding exponential blow-up for many useful cases. Our system model and checking algorithm is not strongly tied to the use of BDDs. In this work, we view our BDD code as simply a package for representing and manipulating Boolean functions symbolically.

COSMOS supports a ternary model of switch-level circuit operation, where X indicates either an unknown or potentially non-digital node voltage. It does this by encoding each node state as a pair of binary values, according to a "dual rail" coding of the circuit state. The excitation function for a node is given by a pair of Boolean expressions specifying how the encoded new state of the node should be computed from the encoded old states [6]. This implementation decision greatly helped the development of our symbolic simulator. When simulating the circuit symbolically, we simply represent the state of each node by pointers to the roots of two BDDs, and compute the new states of a node by evaluating the excitation expressions symbolically.

The following table indicates the performance of our verifier on several different circuits. All CPU times were measured on a DEC 3100 (a 10–20 MIPS machine). We also list the maximum memory requirement of the process, as this is more often the limiting factor in BDD-based symbolic manipulation than is CPU time.

Circuit	Transistors	CPU Time	Memory
64 $\times$ 32 bit moving data stack	16,470	1.25 min.	3.1 MByte
64 $\times$ 32 bit stationary data stack	15,873	7.5 min.	5.7 MByte
1K static RAM	6,875	3.7 min.	9.5 MByte

The two stack examples demonstrate the abstraction capability of our approach. Both started with the same high level specification—defining the behavior of PUSH, POP, and HOLD operations in an abstract stack. We then define how the circuit represents the abstract machine state by trajectory formulas, examples of which will be shown later. The two different circuits represent the abstract state in totally different fashions. In the moving data stack, bit i (counting from the top of stack) of the abstract machine is stored in cell i of the circuit. In the stationary data stack, bit i is stored in location $d - i$ of a static RAM, where d is the current depth of the stack. As a consequence, the symbolic values manipulated in verifying the stationary data stack are more complex than those for the moving data stack. In both cases, however, the performance was acceptable.

The static RAM example indicates just how efficient symbolic verification can be. We have already demonstrated that this class of circuit can be verified by simulating just $O(N \log N)$ scalar patterns [7]. By exploiting the bit-level parallelism of the logical instructions in a conventional machine, we were able to simulate 32 of these patterns at a time. Even so, the symbolic verification outperforms the scalar verification by a factor of 4. Furthermore, creating the assertions for the symbolic verification was far more straightforward than was generating the patterns for the scalar verification. For the scalar verification, we had to consider details of the memory's row and column addressing structure to avoid errors caused by the pessimistic modeling of *X*. No such tuning was required for the symbolic verification.

We are now applying our verifier to more complex circuits such as pipelined data paths and simple microprocessors. We have found the expressive power of our notation and the performance of the verifier acceptable in most of the cases we have considered.

1.4. Related Work

Our approach to verification relates most closely to the symbolic model checking algorithms devised by Bose and Fisher (BF) [2, 3], and by Burch, Clarke, McMillan, and Dill (BCMD) [9]. In fact, all of these approaches are implemented using the same Boolean manipulation code! Furthermore, Bose and Fisher implemented their checker by extending COSMOS. Despite these internal similarities, however, there are significant differences in the capabilities and complexities of the algorithms. In particular, our method is the most restricted in terms of the class of properties that can be verified. Both the BF and BCMD algorithms can decide a class of formulas consisting of a complete branching time, propositional temporal logic under a binary system model. Our method can only be used to verify properties of bounded state sequences. What we loose in expressive power, however, we make up for in computational efficiency. The computational effort required by our model checker is considerably less than theirs. One can view the combined effect of these research projects as providing a spectrum of checking-based verifiers that trade off between expressiveness and performance.

In the BF algorithm, the underlying circuit is assumed to be synchronous and deterministic. Nondeterministic sequential behavior arises due only to the different signals that may be applied to the inputs (expressed as existentially quantified Boolean variables). Their algorithm requires creating an explicit representation of the next state function for every state variable in the system. They create this representation by symbolic simulation. That is, the user identifies each place state is stored in the system, either as charge on a node, or as a pair of complementary values within a static memory element. They then symbolically simulate a single system cycle, where each state variable and each input signal is represented by a distinct Boolean variable. Once they have obtained this Boolean representation of the next state behavior, the validity of a temporal formula can be derived by (rather extensive) symbolic Boolean manipulation. This process of extracting the explicit next state function can be quite costly. In contrast, our method represents the next state function implicitly as a combination of circuit structure and simulation algorithm. We only compute the next state behavior for the particular patterns required to verify a given assertion. These patterns involve far fewer variables than is required by Bose and Fisher's functional extraction.

In the BCMD algorithm, the underlying system can be nondeterministic, and the user can even impose fairness constraints on the model. To support this great expressive power, they require an explicit representation of the next state relation for the entire system. That is, for each state variable in the system they have one Boolean variable representing its "old" value and one representing its "new" value. The next state relation is represented as a Boolean function of all of these variables, where the function yields 1 when the old and new state are related, and 0 otherwise. Generating this characteristic function is even more difficult than generating representations of the individual next state functions for each state variable.

Most other automated approaches to sequential circuit verification are based on testing state machine equivalence [11, 13]. Such methods are useful for comparing two different (but hopefully equivalent) representations of the system, such as one at a register-transfer level and one at a gate level. However, they do not work well for verifying the correctness of incompletely specified systems, nor for reasoning about systems that employ methods, such as pipelining, that shift the sequencing of activities in time. Furthermore, most of these methods assume that the system starts in some known initial state. In actual circuits, the initial state usually cannot be predicted.

Other researchers have suggested symbolic simulation as a means of circuit verification [12, 18]. None of this work has presented a clear methodology for sequential circuit verification, however.

2. Ternary System

Let $\mathcal{B} = \{0, 1\}$ be the set of the binary values and let $\mathcal{T} = \{0, 1, X\}$. The value X is introduced to denote an "unknown", or "don't care" value.

Define the partial order $\sqsubseteq$ on $\mathcal{T}$ as follows: $a \sqsubseteq a$ for all $a \in \mathcal{T}$, $X \sqsubseteq 0$, and $X \sqsubseteq 1$. The

$\cdot_t$	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

$+_t$	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

a	$\overline{a}^t$
0	1
1	0
X	X

Table 1: Ternary Extensions of $\cdot$, $+$, and $\overline{}$.

partial ordering orders values by their "information content." That is, *X* indicates an absence of information while 0 and 1 represent specific, fully-defined values.

We say that ternary values a and b are *compatible*, denoted $a \sim b$, when there is some value $c \in \mathcal{T}$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$. In other words, two scalar values are compatible unless one is 0 and the other is 1.

The *meet*, denoted $\sqcap$, of two ternary values a and b is defined as the largest element $c \in \mathcal{T}$ in the partial order such that $c \sqsubseteq a$ and $c \sqsubseteq b$. For scalar values, if $a = b$, then $a \sqcap b = a$, while if $a \neq b$ then $a \sqcap b = X$. Furthermore, given two compatible ternary values a and b, the *join* between them, denoted $a \sqcup b$, is defined to be the smallest element $c \in \mathcal{T}$ in the partial order such that $a \sqsubseteq c$ and $b \sqsubseteq c$.

It is convenient to define an algebra over $\mathcal{T}$ with operators $\sqcup$, $\sqcap$, as well as the operators $\cdot_t$, $+_t$, and $\overline{}^t$, where the latter are defined in Table 1. These operations are simply extensions of the corresponding Boolean operations $\cdot$ (product), $+$ (sum), and $\overline{}$ (complement).

Let $\mathcal{T}^n$, $n \geq 1$, denote the set of all possible vectors of ternary values of length n, i.e., $\{\langle a_1, \ldots, a_n \rangle | a_i \in \mathcal{T}, 1 \leq i \leq n\}$. The partial order $\sqsubseteq$ is extended to $\mathcal{T}^n$ pointwise: $\vec{a} \sqsubseteq \vec{b}$ iff $a_i \sqsubseteq b_i$ for $1 \leq i \leq n$. Similarly, if $\vec{a} \in \mathcal{T}^n$ and $\vec{b} \in \mathcal{T}^n$, we say $\vec{a} \sim \vec{b}$ iff there exists a vector $\vec{c} \in \mathcal{T}^n$ such that $\vec{a} \sqsubseteq \vec{c}$ and $\vec{b} \sqsubseteq \vec{c}$. In other words, two vectors of scalar ternary values are not compatible when one vector has a 1 in some position and the other vector has a 0 in the same position. Finally, the operations $\sqcup$ and $\sqcap$ are extended pointwise. Note that $\vec{a} \sqcup \vec{b}$ is defined only when $\vec{a} \sim \vec{b}$.

A ternary function, $f\colon \mathcal{T}^n \to \mathcal{T}$, is said to be *monotone* when for any $\vec{a} \in \mathcal{T}^n$ and $\vec{b} \in \mathcal{T}^n$ we have

$$\vec{a} \sqsubseteq \vec{b} \;\Rightarrow\; f(\vec{a}) \sqsubseteq f(\vec{b})$$

This definition is extended pointwise to vector functions, $\vec{f}\colon \mathcal{T}^n \to \mathcal{T}^m$.

The above monotonicity definition is consistent with our use of information content. If a function is monotone, we cannot "gain" any information by reducing the information content of the arguments to the function. In other words, changing some signals from binary values to *X* will either have no effect on the output values, or it will change some binary values to *X*.

To express the behavior of a circuit operating over time, we must reason about *sequences* of states. Conceptually, we will consider the state sequences to be infinite, although the properties we will express can always be determined from some bounded length prefix of the sequence. Define a the set $\mathcal{S}^n$ to consist of all sequences $[\vec{a}_0, \vec{a}_1, \ldots]$ where each $a_i \in \mathcal{T}^n$. The relations $\sqsubseteq$ and $\sim$ are extended from vectors to sequences pointwise. That is, two sequences $[\vec{a}_0, \vec{a}_1, \ldots]$ and $[\vec{b}_0, \vec{b}_1, \ldots]$ are ordered (compatible) if and only if each pair $\vec{a}_i$ and $\vec{b}_i$ is ordered (compatible), for all $i \geq 0$.

For vector $\vec{a}$ and sequence S, the expression $\vec{a}S$ denotes the sequence consisting the vector $\vec{a}$ followed by the vectors in S.

3. Circuit Model

The underlying model of a circuit we use is quite simple, as well as general. A circuit $\mathcal{C}$ is a triple $(\mathcal{N}, \vec{Y}, \mathcal{V})$, where

$\mathcal{N}$: is a set of nodes. Let $s = |\mathcal{N}|$.

$\vec{Y}$: is a vector of excitation functions.

$\mathcal{V}$: is a set of symbolic Boolean variables with which parameterized properties of the circuit are to be expressed.

In the mathematical presentation we will refer to the nodes as $n_1, n_2, \ldots, n_s$, whereas in our examples we often will use more descriptive names. In logic gate circuits the nodes correspond to the primary inputs and the outputs of the gates. In switch-level circuits the nodes correspond to electrical nodes in the circuit.

The excitation functions are defined in a non-traditional way. We view them as expressing "constraints" on the values the nodes can take on one time unit later given the current values on the nodes. By constraint we mean specific binary values, whereas the value *X* indicates that no constraint is imposed. Since the value of an input is controlled by the external environment, the circuit itself does not impose any constraint on the value; hence the excitation of an "input node" is *X*. More formally, if node n_i corresponds to an input to the circuit then $Y_{n_i}(\vec{a}) = X$ for every $\vec{a} \in \mathcal{T}^s$. Nodes that do not correspond to inputs are called *function nodes*. For a function node n_i the excitation function is a monotone ternary function $Y_{n_i}\colon \mathcal{T}^s \to \mathcal{T}$ determined by the circuit topology and functionality. For example, if the current input to a unit delay inverter is 0, then the output of the inverter one time unit later is constrained to be 1.

It should be pointed out that the "time unit" referred to above is the smallest period of time that is distinguishable in the circuit model. The minimum delay in any individual component of the circuit can be significantly larger. Thus we are not limited to unit delay circuit models. For example, by using the transformation technique described in [19], both nominal delay and bounded delay circuit models can be used.

The excitation function for a function node is often a ternary extension of a binary excitation function. Note that such an extension can be done in many ways. However, we require that the extended function agrees with the binary on binary inputs, and that the ternary function obtained is monotone. These requirements simply ensure that we do not "loose" information by extending a binary excitation function to the ternary domain. In other cases, like in switch-level models, the excitation functions derived are already ternary, since even binary signals can generate nondigital voltages on some nodes.

3.1. Circuit Trajectories

State sequences are useful when reasoning about circuit behaviors. However, not all state sequences represent possible behaviors of a circuit. The excitation functions generally restrict the possible state sequences significantly. We formalize this property by introducing the concept of a circuit trajectory. Given a circuit $\mathcal{C}$ and an arbitrary sequence $[\vec{a}_0, \vec{a}_1, \ldots] \in \mathcal{S}^s$ we say that the sequence is a *circuit trajectory* if and only if

$$\vec{Y}(\vec{a}_i) \sqsubseteq \vec{a}_{i+1} \text{ for } i \geq 0.$$

The set of all trajectories of circuit $\mathcal{C}$ is denoted $S(\mathcal{C})$. The above rule for trajectories is consistent with our definition of an excitation function, i.e., a function computing a constraint on the possible value of a node one time unit later. Thus if the current excitation of a node is binary, say a, then the node must take on the value a in the next state in a valid trajectory. On the other hand, if the excitation is *X*, then the node value is not constrained.

On first reading, it may seem strange to define a circuit model where a node with excitation *X* may spontaneously change to either 0 or 1. This is our way of capturing the property that a circuit may exhibit a variety of different behaviors due to variations in the initial state, the primary input values, and the outcome of marginal operating conditions. An input node may change to 0 or 1 on every step, reflecting the fact that its value is determined solely by the operating environment. An uninitialized internal node may change to 0 or 1 to reflect the fact that it could have had either of these as initial values. An internal node that was set to *X* due to marginal operating conditions can change to 0 or 1 to reflect that the actual node state could resolve in either direction.

To illustrate trajectories, consider a unit delay inverter. Assume the circuit contains two nodes: the inverter input n_1 and the inverter output n_2. The excitation functions are $\vec{Y}(\vec{a}) = \langle X, \overline{a_1}^t \rangle$. It is easy to accept that, for example, $\langle 0, 1\rangle, \langle 0, 1\rangle, \langle 0, 1\rangle, \ldots$ and $\langle 0, 1\rangle, \langle 1, 1\rangle, \langle 1, 0\rangle, \langle 1, 0\rangle, \ldots$ are valid circuit trajectories. Less intuitive is that a sequence like $\langle X, 0\rangle, \langle X, 1\rangle, \langle X, 0\rangle, \langle X, 1\rangle, \ldots$ is also a valid circuit trajectory. Our methodology verifies properties for all circuit trajectories that satisfies some conditions, including degenerate trajectories as this one. Consequently it will perform a stronger verification than absolutely necessary. Again, this pessimism stems from our use of *X* as a completely unknown value.

4. Specification Language

Our specification language describes a property of the circuit as an *assertion* of the form $[A \implies C]$, where both A and C are *symbolic trajectory formulas* expressing constraints on the circuit trajectory.

4.1. Symbolic Expressions

Before we can define our language, we need to introduce some notation and definitions. If $\mathcal{V}$ is a set of symbolic Boolean variables then an *interpretation*, ϕ, is a function $\phi\colon \mathcal{V} \rightarrow \mathcal{B}$ assigning a binary value to each variable. Let Φ be the set of all possible interpretations, i.e., $\Phi = \{\phi\colon \mathcal{V} \rightarrow \mathcal{B}\}$. A *domain constraint*, $\mathcal{D} \subseteq \Phi$, defines a restriction on the values assigned to the variables. We will denote such domain constraints by Boolean expressions. That is, let E be a Boolean expression over elements of $\mathcal{V}$.[1] This expression defines a Boolean function $e\colon \Phi \rightarrow \mathcal{B}$ and thus denotes the domain constraint $\mathcal{D} = \{\phi | e(\phi) = 1\}$. The set of all interpretations Φ is denoted by the Boolean function **1**, defined as yielding 1 for all interpretations. Expressing domain constraints by Boolean expressions allows us to compactly specify many different circuit operating conditions with a single formula.

4.2. Symbolic Trajectory Formulas

A trajectory formula expresses a set of constraints on a circuit trajectory. When the formula contains Boolean expressions, each interpretation of the variables yields a different set of constraints.

A *step-level* symbolic trajectory formula is defined recursively as:

1. **Constants**: TRUE is a trajectory formula.
2. **Atomic propositions**: for $n_i \in \mathcal{N}$ the following are trajectory formulas:
   (a) $(n_i = 1)$
   (b) $(n_i = 0)$.
3. **Conjunction**: $(F_1 \wedge F_2)$ is a trajectory formula if F_1 and F_2 are trajectory formulas.
4. **Domain restriction**: $(E \rightarrow F)$ is a trajectory formula if E is a Boolean expression over $\mathcal{V}$ and F is a trajectory formula.
5. **Next time**: $(\mathbf{X}_s F)$ is a trajectory formula if F is a trajectory formula.

We say that a formula is *instantaneous* when it does not contain any next time operator $\mathbf{X}_s$. For convenience, we often drop parentheses when the intended precedence is clear.

The truth of a formula F is defined relative to a circuit, an interpretation ϕ of the variables in $\mathcal{V}$, and a circuit trajectory. The truth of F, written $\mathcal{C}, \phi, S \models F$, is defined recursively. In the following, assume that both S and $\vec{a}S$ are trajectories of $\mathcal{C}$.

1. $\mathcal{C}, \phi, S \models$ TRUE holds trivially.

[1]For the sake of brevity, we omit a formal syntax of Boolean expressions. Any standard expression syntax suffices.

2. (a) $\mathcal{C}, \phi, \vec{a}S \models (n_i = 1)$ iff $a_i = 1$.
   (b) $\mathcal{C}, \phi, \vec{a}S \models (n_i = 0)$ iff $a_i = 0$.

3. $\mathcal{C}, \phi, S \models (F_1 \wedge F_2)$ iff $\mathcal{C}, \phi, S \models F_1$ and $\mathcal{C}, \phi, S \models F_2$

4. $\mathcal{C}, \phi, S \models (E \rightarrow F)$ iff $e(\phi) = 0$ or $\mathcal{C}, \phi, S \models F$, where e is the Boolean function denoted by the Boolean expression E.

5. $\mathcal{C}, \phi, \vec{a}S \models \mathbf{X}_s F$ iff $\mathcal{C}, \phi, S \models F$.

For an instantaneous formula, its truth can be defined relative to a single state. For instantaneous formula F, the notation $\mathcal{C}, \phi, \vec{a} \models F$ indicates that F holds under interpretation ϕ for state $\vec{a}$. A formal definition of this notation can be derived by a straightforward adaptation of rules 1–4 above.

4.3. Assertions

Our verification methodology entails proving *assertions* about the model structure. These assertions are of the form $[A \implies C]$, where the *antecedent* A and the *consequent* C are trajectory formulas. The truth of an assertion is defined relative to a circuit $\mathcal{C}$ and an interpretation ϕ. Unlike a formula, however, an assertion is considered true only if it holds for all trajectories. That is, $\mathcal{C}, \phi \models [A \implies C]$, when for every $S \in \mathcal{S}(\mathcal{C})$ we have that $\mathcal{C}, \phi, S \models A$ implies that $\mathcal{C}, \phi, S \models C$. Given a circuit and an assertion, the task of our checking algorithm is to compute the Boolean function expressing the set of interpretations under which the assertion is true. For most verification problems, this should simply be the constant function **1**, i.e., the assertion should hold under all variable interpretations.

4.4. Properties

We have intentionally chosen to introduce only a heavily restricted trajectory formula syntax for our base logic. As discussed previously, we do not support more sophisticated temporal operators such as “Until,” “Globally,” or “Eventually.” Furthermore, we do not even permit such elementary logic operators such as disjunction, negation, or quantification.

By imposing these restrictions, we can guarantee the following key property:

Proposition 1 *For any trajectory formula F, and any interpretation ϕ, one of the following cases must hold:*

1. *There is no trajectory $S \in \mathcal{S}(\mathcal{C})$ for which $\mathcal{C}, \phi, S \models F$, or*

2. *There exists a unique trajectory $S_{F,\phi} \in \mathcal{S}(\mathcal{C})$ such that for every $S \in \mathcal{S}(\mathcal{C})$ we have $\mathcal{C}, \phi, S \models F$ if and only if $S_{F,\phi} \sqsubseteq S$.*

In the first case above, we say that the formula F is not satisfiable under interpretation ϕ. In the second case, we refer to the sequence $S_{F,\phi}$ as the *weakest trajectory* satisfying formula F under interpretation ϕ.

This theorem can be proved by induction on the formula structure. Rather than prove it here, however, we will show in a later section how the checking algorithm can compute a Boolean function describing the set of interpretations for which the formula is satisfiable, as well as a symbolic sequence representing the weakest trajectory for every satisfiable interpretation.

Note that this theorem expresses a very strong property of our logic. It demonstrates the reason why we can verify an assertion by simulating a single symbolic sequence, namely the one encoding the weakest trajectories allowed by the antecedent for every interpretation. It is stronger than the simple monotonicity condition that if $\mathcal{C}, \phi, S \models F$ and $S \sqsubseteq S'$, then $\mathcal{C}, \phi, S' \models F$.

Observe that this property would not hold if our formula syntax permitted disjunction. For example, given a circuit with two nodes n_1 and n_2, the two formulas $n_1 = 1$ and $n_2 = 0$ have weakest trajectories where the first elements are the vectors $\langle 1, X\rangle$ and $\langle X, 0\rangle$, respectively. Furthermore, the formula $n_1 = 1 \wedge n_2 = 0$ has a weakest trajectory where the first element is the vector $\langle 1, 0\rangle$. On the other hand, there is no weakest trajectory satisfying $n_1 = 1 \vee n_2 = 0$. Only the vector $\langle X, X\rangle$ is less than both $\langle 1, X\rangle$ and $\langle X, 0\rangle$, and a trajectory having this as its initial vector satisfies neither $n_1 = 1$ nor $n_2 = 0$.

If our syntax permitted negation, then we would lose even the monotonicity property of our logic. For example, in the circuit described above, the formula $\neg(n_1 = 1)$ would be satisfied by a trajectory having first element $\langle X, X\rangle$, but not by one having first element $\langle 0, X\rangle$.

Disjunction is a useful extension to our language and can be implemented using quantified Boolean variables. Negation, on the other hand, seems to run contrary to the principles of ternary modeling.

5. Extensions

The logic, as described above, is convenient for deriving the underlying theory. Unfortunately, expressing "interesting" assertions about real circuits using only the constructs above is very tedious. Two shortcomings make using the logic cumbersome: the fine granularity of the timing, and the lack of more powerful logical constructs. It is convenient to add extensions that do not add any expressive power, but make it easier to write assertions.

This basic structure of starting with a minimal basic logic and then adding more elaborate structures as extensions also mirrors our current implementation. The implementation consists of two parts. The underlying logic, with some few extensions, is taken care of by our modified version of the COSMOS symbolic switch-level simulator. The syntactic extensions are supported by a front-end written in SCHEME. The user writes SCHEME code that, when evaluated, generates a file of low-level simulation commands which are then evaluated by the simulator.

5.1. Timing Extensions

Our main interest is in verifying synchronous circuits. Hence, we would like to reason about time on a more abstract level than basic time units. For many VLSI circuits it is natural to describe the desired behavior in terms of phase level behavior. Let a *phase* be a period of time during which no inputs (including clocks) change. For simplicity, we will assume that all phases have the same duration. For example, if phases are k basic time units long and we want the instantaneous formula F to hold for the entire phase, then this can be translated into

$$\bigwedge_{j=0}^{k-1} (\mathbf{X}_s^j F)$$

where, for $j \geq 0$, $\mathbf{X}_s^j$ denotes j repetitions of the next time operator, and $\mathbf{X}_s^0 F$ is defined as F. A "next phase" operator $\mathbf{X}_p$ can then be defined simply as $\mathbf{X}_s^k$. Thus a formula in a logic based on the next-phase operator can be translated directly into a formula in the logic based on the next-step operator. In our current implementation, the logic we support only allows the next-phase operator. The transformation into the next-step logic is taken care of by the COSMOS simulator so that event-scheduling can be used to achieve good performance.

If we are using a nominal or bounded delay circuit model, the basic time step corresponds directly with some exact amount of time. Hence, in these cases it is easy to define the length of a phase. At other times we are not interested in the exact timing of the circuit. For example, if we do not have delay values available (pre-layout) or if we use a very inexact delay model (say unit-delay) it is somewhat meaningless to say that a phase is k time units long. In these situations we often want to express the behavior in terms of "stable" phases. In other words, a phase may defined to be as long as needed for the circuit to stabilize. Of course, we must also take care of the case where the circuit never stabilizes, but oscillates. The solution to this is to "force" the circuit simulator to stabilize after some reasonably large number of steps. This can be accomplished by running a normal simulation for that many steps and then forcing oscillating nodes to X. We leave the details on how to do this symbolically to the interested reader.

5.2. Logic Extensions

As already alluded to above, it is very tedious to write assertions using only the basic logic. Thus in our SCHEME translator we provide a number of syntactic extensions. In the following we will outline some of these.

An obvious extension to the logic is to allow the user to write $n_i = E$, where $n_i \in \mathcal{N}$ and E is some Boolean expression over $\mathcal{V}$. The meaning of this is simply $[E \rightarrow (n_i = 1)] \wedge [\overline{E} \rightarrow (n_i = 0)]$, where $\overline{E}$ denotes the Boolean complement of E.

Our next addition is to allow the user to specify a domain restriction for a complete assertion, i.e., we allow the user to write $E \rightarrow [A \implies C]$, meaning that the assertion only needs to hold for interpretations $\phi \in \Phi$ such that $e(\phi) = 1$, where e is the Boolean function denoted by E. This is simply a short-hand for the assertion $[(E \rightarrow A) \implies (E \rightarrow C)]$.

The following extension to the logic, dealing with finite integer domains, is perhaps the most useful of them all. Let $\vec{E}$ be a vector of Boolean expressions over $\mathcal{V}$. Given an interpretation $\phi \in \Phi$ one can view $\vec{E}$ as the binary encoding of a number. Denote this number as $|\vec{E}(\phi)|$. Let $\vec{v}$ denote a vector of Boolean expressions over $\mathcal{V}$. What we would like to do is to use $\vec{E}$ to select one of the Boolean expressions in $\vec{v}$. Assume the vector contains m expressions. The *value* of $v_{\vec{E}}$ is defined as:

$$v_{\vec{E}} \;=\; (|\vec{E}| = 0) \cdot v_0 \;+\; \ldots \;+\; (|\vec{E}| = m-1) \cdot v_{m-1}$$

where if $\vec{E} = \langle E_{p-1}, E_{p-2}, \ldots, E_0 \rangle$, and $\langle i_{p-1}, i_{p-2}, \ldots, i_0 \rangle$ is the binary encoding of i, the expression $(|\vec{E}| = i)$ is the Boolean expression $\prod_{0 \le j < p}((E_j \cdot i_j) + (\overline{E_j} \cdot \overline{i_j}))$. It is easy to see that given an interpretation $\phi \in \Phi$ this definition captures our intuition of what it means to select one of the expressions.

Finally, we would also like to do a similar selection among circuit nodes. Let $\vec{\mathsf{N}}$ denote a vector of m circuit nodes. The expression $\mathsf{N}_{\vec{E}} = G$, where G is some Boolean expression over $\mathcal{V}$, is simply a shorthand for the trajectory formula

$$[(|\vec{E}| = 0) \rightarrow (\mathsf{N}_0 = G)] \wedge [(|\vec{E}| = 1) \rightarrow (\mathsf{N}_1 = G)] \wedge \ldots \wedge [(|\vec{E}| = m-1) \rightarrow (\mathsf{N}_{m-1} = G)]$$

where $(|\vec{E}| = i)$ is defined as above.

Given that the front-end is embedded in SCHEME, and the user actually writes SCHEME code, it is easy to define new extensions. In fact, by writing SCHEME procedures it becomes very natural to express the circuit assertions in a hierarchical way, improving the readability of—and consequently the confidence in—the assertions.

6. Examples

The extensions described in the previous section provide a phase level timing model and an assertion language with more powerful abstraction capabilities. Given these extensions we claim the logic is both sufficiently powerful and easy to use to express a wide variety of verification tasks. We support this claim with some illustrative examples.

6.1. Moving Data Stack

First we will demonstrate the utility of the temporal notation in expressing phase level circuit timing and pipelining. The example is the nMOS stack circuit described by Mead and Conway [16]. Figure 1 shows the block diagram for the circuit. The circuit operates with a two-phase nonoverlapping clock. The stack command is specified by a pair of signals, OP1 and OP2, which are multiplexed onto a single circuit input `Op`. The control signal is pipelined half a clock cycle ahead of the data signals. That is, the command for clock cycle t is specified by supplying OP1 on the `Phi2` phase of cycle $t-1$, and OP2 on the `Phi1` phase of cycle t, as illustrated in Figure 2. The input data must be supplied during the `Phi1` phase, and the output is valid during the `Phi2` phase. For each stack cell, we will consider the "state" of the cell to be the value held there during the `Phi1` phase of the cycle.

Omitted Figure: stack-block

Figure 1: **Block Diagram of Mead and Conway Stack**. Bit i of the stack is stored as charge on node $\texttt{Cl}_i$ (negative logic).

Omitted Figure: stack-timing

Figure 2: **Timing Diagram Stack**. The two control signals for a given clock cycle are multiplexed on the single line `Op`.

To illustrate a possible assertion for this circuit, consider checking that a PUSH operation can be carried out correctly. The PUSH operation is performed when OP1 $= 1$ and OP1 $= 0$. Informally, a PUSH operation should put the pushed value into cell 0 and move the contents of cell i to cell $i+1$ for every i less than $k-1$, where k is the number of cells in the circuit. For convenience, let $\vec{\texttt{Cl}}$ be a vector of the $\texttt{Cl}_i$ nodes. Let u and v be two Boolean variables, and $\vec{I} = \langle I_{p-1}, \ldots, I_0 \rangle$ be a vector of Boolean variables, where $k \leq 2^p$.

First, we must include in our assertions a specification of how the clocks should be operated. We use the shorthand *Clock1* to denote the trajectory formula

$$(\texttt{Phi1} = 1) \wedge \mathbf{X}_p(\texttt{Phi1} = 0) \wedge \mathbf{X}_p^2(\texttt{Phi1} = 0) \wedge \mathbf{X}_p^3(\texttt{Phi1} = 0),$$

Clock2 to denote

$$(\texttt{Phi2} = 0) \wedge \mathbf{X}_p(\texttt{Phi2} = 0) \wedge \mathbf{X}_p^2(\texttt{Phi2} = 1) \wedge \mathbf{X}_p^3(\texttt{Phi2} = 0),$$

and *Cycles* to denote

$$\textit{Clock1} \wedge \textit{Clock2} \wedge \mathbf{X}_p^4(\textit{Clock1} \wedge \textit{Clock2}) \wedge \mathbf{X}_p^8[(\texttt{Phi1} = 1) \wedge \texttt{Phi2} = 0].$$

The formula *Cycles* defines the operation of the clocks over the time period illustrated in Figure 2.

Since negative logic is used internally, let $\textit{Stored}(\vec{I}, u)$ be a shorthand for $\texttt{Cl}_{\vec{I}} = \overline{u}$. Let *Push* denote $\mathbf{X}_p^2(\texttt{OP} = 1) \wedge \mathbf{X}_p^4(\texttt{OP} = 0)$.

We can now express our assertion as:

$$\begin{aligned}&(\vec{I} < k-1) \rightarrow \\ &\quad \Big[\textit{Cycles} \wedge \textit{Push} \wedge \mathbf{X}_p^4[(\texttt{In} = v) \wedge \textit{Stored}(\vec{I}, u)] \implies \mathbf{X}_p^8[\textit{Stored}(0, v) \wedge \textit{Stored}(\vec{I}+1, u)]\Big].\end{aligned}$$

Note that $(\vec{I} < k-1)$ denotes a Boolean function over $\mathcal{V}$ that, for a given interpretation ϕ, is 1 iff $|\vec{I}(\phi)| < k-1$. Similarly, $\vec{I}+1$ denotes a vector of Boolean functions such that, for a given interpretation ϕ, $|(\vec{I}+1)(\phi)| = |\vec{I}(\phi)| + 1$.

In this example, verifying the correct operation of a single stack operation requires reasoning about the behavior of the circuit over parts of 3 clock cycles. Such is often the case in pipelined systems.

6.2. Static RAM

Our second example is a static RAM, as illustrated in Fig. 3. The data for each memory location i are stored on node $\texttt{B}_i$. Within the RAM cell there is a node holding the complementary value. Unlike in the verification described in [7] based on scalar simulation, we do not need to explicitly set or check this complementary value. The phase level timing abstraction handles this automatically. That is, asserting a value on node $\texttt{B}_i$ for 2 unit steps forces the complementary value onto the other node in the RAM cell.

Omitted Figure: ram-block

Figure 3: **Block diagram of Static RAM.** State is stored in location i as a pair of complementary values on nodes $\texttt{Ch}_i$ and $\texttt{Cl}_i$.

The only information we need to know about the design of the RAM is the names of the input/output signals, and the names of the storage nodes of each memory cell, as well as the interface timing. For ease of exposition, we express the specification at the *clock cycle* level in terms of a cycle-level next-time operator $\mathbf{X}_c$. Given a specification of the clocking patterns and interface timing for the circuit, we could translate these assertions into phase-level assertions in a similar fashion as we did for the moving data stack.

It is easy to convince oneself that a correct memory must satisfy three properties: 1) that we can write successfully into each cell i, 2) that we can nondestructively read the content of each cell i, and 3) that, unless we are writing into some cell i, the value stored in cell i should not change. Let $\vec{\texttt{B}}$ be a vector of the $\texttt{B}_i$ nodes. Furthermore, let u, v, and w be Boolean variables, and $\vec{I} = \langle I_{p-1}, \ldots, I_0 \rangle$ and $\vec{J} = \langle J_{p-1}, \ldots, J_0 \rangle$ be vectors of Boolean variables, where p is the width of the address input. We use the shorthand *Operate*$(\vec{I}, u, w)$ to denote the formula

$$(\texttt{write} = w) \wedge (\texttt{Din} = u) \wedge (\texttt{A}_{p-1} = I_{p-1}) \wedge \cdots \wedge (\texttt{A}_0 = I_0).$$

Similarly, let *Stored*$(\vec{J}, v)$ denote the formula $(\texttt{B}_{\vec{I}} = v)$. With this notation, we can express all three conditions mentioned above with a single assertion.

$$\begin{aligned}
&\textit{Stored}(\vec{J}, v) \wedge \textit{Operate}(\vec{I}, u, w) \\
&\quad \Longrightarrow \ \mathbf{X}_c \ \Big([w \rightarrow \textit{Stored}(\vec{I}, u)] \quad \wedge \\
&\qquad\qquad [\overline{w} \cdot (\vec{I} = \vec{J}) \rightarrow (\text{DOUT} = v)] \quad \wedge \\
&\qquad\qquad [\overline{w} + (\vec{I} \neq \vec{J}) \rightarrow \textit{Stored}(\vec{J}, v)]\Big)
\end{aligned}$$

where $\vec{I} \neq \vec{J}$ denotes the Boolean expression $(I_{p-1} \oplus J_{p-1}) + \cdots + (I_0 \oplus J_0)$, and $\vec{I} = \vec{J}$ denotes the complement of this expression.

6.3. Discussion

As these examples illustrate, our notation works well for expressing the state transition properties of a circuit. For circuits, such as memories and data paths, this is a fairly natural form of specification. That is, one thinks of each operation of the system as updating some portion of the stored system state.

In these examples, we constructed the specification assertion in a hierarchical way, starting with low level timing information and working up to more abstract system operations. This seems to be a fairly convenient way to view the system at different abstraction levels. Because we actually write our specifications as SCHEME programs, we can use the procedural abstraction capabilities of SCHEME to express the specification hierarchically.

7. Symbolic Simulation

As we have shown, symbolic formulas provide a concise means to specify desired properties of the circuit under many different operating conditions. We are now ready to introduce a method of verifying the assertions via symbolic simulation. The key idea is to preserve the symbolic structure of the formulas in the verification algorithm. By doing so, we can replace the need for large amounts of case analysis with algebraic manipulation.

7.1. Symbolic Algebras

In creating a symbolic model, we extend the scalar model defined in terms of the binary and ternary domains $\mathcal{B}$ and $\mathcal{T}$, to one defined in terms of binary- and ternary-valued functions over the variables $\mathcal{V}$. Define the symbolic domain $\mathcal{B}(\mathcal{V})$ (respectively, $\mathcal{T}(\mathcal{V})$) as denoting the set of functions mapping an interpretations in Φ to $\mathcal{B}$ (resp., $\mathcal{T}$). More formally

$$\mathcal{B}(\mathcal{V}) = \{f\colon \Phi \rightarrow \mathcal{B}\}$$

and

$$\mathcal{T}(\mathcal{V}) = \{f\colon \Phi \rightarrow \mathcal{T}\}.$$

We then extend the operations defined over scalar values to create a symbolic algebra.

We can also extend the vector and sequence algebra defined over scalar values to their counterparts defined over symbolic values. That is, define the vector domain $\mathcal{T}(\mathcal{V})^n$ as

$$\mathcal{T}(\mathcal{V})^n \quad = \quad \{\langle a_1, \ldots, a_n\rangle | a_i \in \mathcal{T}(\mathcal{V})\}.$$

In implementing a symbolic simulator, we in effect extend the excitation function $\vec{Y}$ to the symbolic domain as $\vec{Y}\colon \mathcal{T}(\mathcal{V})^s \rightarrow \mathcal{T}(\mathcal{V})^s$. For $\vec{a} \in \mathcal{T}(\mathcal{V})^n$, let $\vec{a}(\phi) \in \mathcal{T}^n$ denote the vector with each element i equal to $a_i(\phi)$. In this way, we can view the symbolic vector $\vec{a} \in \mathcal{T}(\mathcal{V})^n$ either as a vector of symbolic elements, or as a symbolic value which for a given interpretation yields a scalar vector.

We extend most operations from scalar to symbolic domains in a uniform way. Consider an operation $op\colon \mathcal{D}_1 \times \mathcal{D}_2 \rightarrow \mathcal{D}_3$, defined over vectors, single elements, or a combination of the two. Its symbolic counterpart $op\colon \mathcal{D}_1(\mathcal{V}) \times \mathcal{D}_2(\mathcal{V}) \rightarrow \mathcal{D}_3(\mathcal{V})$ is defined such that for all $a \in \mathcal{D}_1(\mathcal{V})$ and $b \in \mathcal{D}_2(\mathcal{V})$, we have $(a\ op\ b)(\phi) = a(\phi)\ op\ b(\phi)$. We use this method to extend the ternary algebraic operations $\cdot_t$, $+_t$, and $\overline{}^t$, as well as the operation $\sqcap$.

When extending a relation R symbolically, we define the result to be a function specifying the interpretations under which its arguments are related. That is, given a binary relation $R \subseteq \mathcal{D}_1 \times \mathcal{D}_2$, define $R\colon \mathcal{D}_1(\mathcal{V}) \times \mathcal{D}_2(\mathcal{V}) \rightarrow \mathcal{B}(\mathcal{V})$ as $(a\ R\ b)(\phi) = 1$ if and only $a(\phi)\ R\ b(\phi)$. We use this method to define operations $\sim$ and $\sqsubseteq$ over both single elements and vectors.

7.2. Special Operations

We require one operation that is extended to vectors in a nonstandard way. Define the infix operator ?: $\mathcal{B} \times \mathcal{T} \rightarrow \mathcal{T}$ as a ? b equals b if a is 1, and equals X otherwise. When extending this operation to vectors, only the second argument is vector-valued. That is the operation ?: $\mathcal{B} \times \mathcal{T}^n \rightarrow \mathcal{T}^n$ is defined as $(a \ ?\ \vec{b})_i = a\ ?\ b_i$. This operation is then extended symbolically in the manner described above.

As a final operation, we define a variant of the join operation that is defined even when for some $\phi \in \Phi$, we have $\vec{a}(\phi) \not\sim \vec{b}(\phi)$. When using this operation, we will separately keep track of the conditions under which the arguments are compatible. Define the operation $\overset{*}{\sqcup}\colon \mathcal{T}(\mathcal{V})^n \times \mathcal{T}(\mathcal{V})^n \rightarrow \mathcal{T}(\mathcal{V})^n$ as

$$(\vec{a} \overset{*}{\sqcup} \vec{b})(\phi) \quad = \quad \begin{cases} \vec{a}(\phi) \sqcup \vec{b}(\phi), & \vec{a}(\phi) \sim \vec{b}(\phi) \\ \vec{X}, & \text{otherwise} \end{cases}$$

where $\vec{X}$ denotes a vector with all elements equal to X.

7.3. Translating Instantaneous Formulas to Symbolic Vectors

Given the above definitions, we first give a procedure that for an instantaneous formula F derives a Boolean function OK_F and a vector $\vec{a}_F$. The function OK_F can be viewed as a "domain" function of F in the sense that $OK_F(\phi) = 1$ iff under this interpretation F can hold true in some state. Furthermore, we will also show that $\vec{a}_F$ is "weakest" in the sense that if $OK_F(\phi) = 1$, then $\vec{a}_F(\phi) \sqsubseteq \vec{b}$ for every state vector $\vec{b} \in \mathcal{T}^s$ for which F holds under this interpretation. The function OK_F and the vector $\vec{a}_F$ are defined recursively as:

1. If F is TRUE then $OK_F = 1$, and $\vec{a}_F = \langle X, \ldots, X \rangle$.
2. (a) If F is $(n_i = 1)$ then $OK_F = 1$, and $\vec{a}_F = \langle X, \ldots, X, 1, X, \ldots, X \rangle$, where the 1 is in position i.
   (b) If F is $(n_i = 0)$ then $OK_F = 1$, and $\vec{a}_F = \langle X, \ldots, X, 0, X, \ldots, X \rangle$, where the 0 is in position i.

3. If F is $(F_1 \wedge F_2)$ then $OK_F = OK_{F_1} \cdot OK_{F_2} \cdot (\vec{a}_{F_1} \sim \vec{a}_{F_2})$, and $\vec{a}_F = \vec{a}_{F_1} \overset{*}{\sqcup} \vec{a}_{F_2}$.

4. If F is $(E \rightarrow F_1)$ then $OK_F = \overline{e} + OK_{F_1}$, and $\vec{a}_F = e \;?\; \vec{a}_{F_1}$, where e is the Boolean function denoted by the expression E.

Proposition 2 *Given a circuit $\mathcal{C}$, let F be an instantaneous formula and OK_F and $\vec{a}_F$ be derived as above. Then $OK_F(\phi) = 1$ iff there exists some state $\vec{b} \in \mathcal{T}^s$ such that $\mathcal{C}, \phi, \vec{b} \models F$. Furthermore, if $OK_F(\phi) = 1$, then $\mathcal{C}, \phi, \vec{b} \models F$ iff $\vec{a}_F(\phi) \sqsubseteq \vec{b}$.*

This proposition can be proved by induction on the formula structure.

7.4. Checking Assertions

Our first step in verifying an assertion is to rewrite the antecedent and consequent into a normal form where all next-time operators are collected together. It is easy to show that a trajectory formula F can be rewritten into $F_0 \wedge \mathbf{X}_s F_1 \wedge \mathbf{X}_s^2 F_2 \wedge \ldots \wedge \mathbf{X}_s^{k-1} F_{k-1}$, for some $k \geq 1$, where each F_i is instantaneous. Note that some of the F_i's might be the trivial formula TRUE. Note also that such a sequence can be extended by appending $\mathbf{X}_s^i$TRUE for $i \geq k$. Hence, without any loss of generality, we will henceforth assume that the antecedent and the consequent in an assertion are trajectory formulas in normal form containing the same number of terms.

Given an assertion $[A \implies C]$ of the form

$$\left[A_0 \wedge \mathbf{X}_s A_1 \wedge \ldots \wedge \mathbf{X}_s^{k-1} A_{k-1} \implies C_0 \wedge \mathbf{X}_s C_1 \wedge \ldots \wedge \mathbf{X}_s^{k-1} C_{k-1}\right]$$

define a sequence of symbolic ternary vectors $\vec{x}_0, \ldots, \vec{x}_{k-1}$ as follows:

$$\vec{x}_i = \begin{cases} \vec{a}_{A_0}, & i = 0 \\ \vec{Y}(\vec{x}_{i-1}) \overset{*}{\sqcup} \vec{a}_{A_i}, & i > 0. \end{cases}$$

Define the Boolean function $OK_A = \prod_{0 \leq i < k} OK_{A_i}$, where $\prod$ denotes Boolean product. This function yields 0 for those interpretations for which the antecedent contains some internal inconsistency. For example, the formula $A = (n_i = a) \wedge (n_i = b)$ would have $OK_A = \overline{a \oplus b}$, because this formula cannot be satisfied when $\phi(a) \neq \phi(b)$. Define the Boolean function $Traj = \prod_{1 \leq i < k} [\vec{Y}(\vec{x}_{i-1}) \sim \vec{a}_{A_i}]$. This function yields 0 for those interpretations where an incompatibility arises in the trajectory.

We can show that A is satisfiable under some interpretation ϕ if and only if $OK_A(\phi) \cdot Traj(\phi) = 1$. Furthermore, we can extend the sequence $\vec{x}_0, \ldots, \vec{x}_{k-1}$ to be an infinite sequence by defining $\vec{x}_i = \vec{Y}(\vec{x}_{i-1})$ for all $i \geq k$. It can then be shown that for interpretation ϕ the sequence $\vec{x}_0(\phi), \vec{x}_1(\phi), \ldots$ is the weakest trajectory satisfying A under interpretation ϕ. This construction then provides a proof of Proposition 1. This demonstrates how our symbolic simulator can set up the weakest allowable conditions allowed by the antecedent under all possible interpretations.

To check the consequent, define the Boolean function $OK_C = \prod_{0 \leq i < k} OK_{C_i}$. This function yields 0 for those interpretations for which the consequent contains some internal inconsistency.

Finally, define the Boolean function $\mathit{Check} = \prod_{0 \leq i < k} [\vec{a}_{C_i} \sqsubseteq \vec{x}_i]$. This function yields 0 for those interpretations where some trajectory satisfying the antecedent may violate the consequent.

Now define $\mathit{OK}_{[A \implies C]}$ as:

$$\overline{\mathit{OK}_A} + \overline{\mathit{Traj}} + (\mathit{OK}_C \cdot \mathit{Check})$$

Informally, this equation states that the assertion is true under those interpretations for which the antecedent is unsatisfiable (due either to internal inconsistencies or to an incompatibility in the trajectory), as well as those for which the consequent holds (i.e, it is both internally consistent and is satisfied.)

The main result of this paper is captured in the following theorem:

Theorem 1 *Given a circuit $\mathcal{C}$ and an assertion $[A \implies C]$ let $\mathit{OK}_{[A \implies C]} \in \mathcal{B}(\mathcal{V})$ be derived as above. Then*

$$\mathcal{C}, \phi \models [A \implies C] \quad \textit{if and only if} \quad \mathit{OK}_{[A \implies C]}(\phi) = 1.$$

Hence, determining whether a circuit satisfies $[A \implies C]$ is reduced to determining whether $\mathit{OK}_{[A \implies C]} = 1$.

7.5. Summary of Checking Algorithm

In practice, we encode the ternary values as pairs of Boolean values. Thus we express the ternary functions over $\mathcal{V}$ as pairs of Boolean functions over $\mathcal{V}$. It is straightforward to implement all of the operations defined above using such an encoding. Thus, to verify an assertion $[A \implies C]$ we would proceed as follows: First convert A and C into normal forms containing the same number of terms. Now compute the sequence of states $\vec{x}_i$ as defined above. Although the above presentation used $k-1$ distinct vectors, the implementation need only retain the most recent vector. In fact, the vector can be viewed as the current circuit state. Hence, step i in this verification would entail: 1) run the simulator one "step" to derive $\vec{Y}(\vec{x}_{i-1})$, 2) compute OK_{A_i} and $\vec{a}_{A_i}$, 3) compute $(\vec{Y}(\vec{x}_{i-1}) \sim \vec{a}_{A_i})$, 4) compute $\vec{x}_i$, 5) compute $\vec{a}_{C_i}$, and 6) compute $(\vec{a}_{C_i} \sqsubseteq \vec{x}_i)$. As we go along, we also maintain Boolean functions representing the 4 constituent components of the expression for $\mathit{OK}_{[A \implies C]}$ shown above. In the end we simply combine these together and check that the derived function is the constant function **1**. If the test fails, the derived function provides diagnostic information indicating which cases encounter difficulties.

7.6. Example

Consider a 4-bit memory of the structure illustrated in Figure 3. In verifying the correctness of a READ operation, one aspect of the desired behavior is that after reading with the address inputs set to i, the valued stored in memory location i should appear on the output. This can be expressed by the assertion (in our extended syntax):

$$\left[\mathit{Stored}(\vec{I}, u) \wedge (\texttt{write} = 0) \wedge (\vec{\texttt{A}} = \vec{I}) \implies \mathbf{X}_c(\texttt{Dout} = u)\right]$$

Omitted Figure: sram-pattern

Figure 4: **Verification of 4-bit RAM Read Operation.**

Omitted Figure: pseudo-xor

Figure 5: **Pseudo XOR Circuit**. This circuit satisfies the assertion for an XOR circuit, because the antecedent trajectory fails whenever the output should be nonzero.

The antecedent is converted into the simulation pattern outlined on the left hand side of Figure 4. That is, the address inputs are set to Boolean variables encoding the possible addresses. The cells are set to more complex symbolic values. For each possible interpretation of the address variables, we see that one cell is set to value u, while the other three cells are set to X. Verifying the assertion involves simulating this single pattern and then checking that the resulting value on `Dout` equals u.

This example illustrates how our methodology combines symbolic and ternary values when encoding the different possible circuit states. In a binary model of the circuit, there are 16 (in general 2^k) different initial binary states of the memory. These cases can be covered by 8 (in general $2k$) different initial states in a ternary model—each having one cell set to 0 or 1 and the rest set to X. These different cases can then be encoded by a single symbolic case in terms of 3 (in general $1 + \log k$) Boolean variables. As the size of the memory k grows, the reduction in complexity we realize becomes dramatic.

8. Areas for Future Investigation

The checker described here provides a useful tool for reasoning about digital circuits. However, some questions remain as to exactly how it can and should be used. In particular, we must still devise a comprehensive theory on how to prove that a circuit fully realizes its specification.

To illustrate one subtlety of this task, consider trying to show that a circuit with inputs A and B and output Out implements the exclusive-or function. Intuitively, it seems that it would be sufficient to prove that circuit obeys the assertion $[(\mathtt{A} = a) \wedge (\mathtt{B} = b) \implies \mathbf{X}_s(\mathtt{Out} = a \oplus b)]$. Unfortunately, this is not the case. For example, this assertion is obeyed by the rather useless circuit of Figure 5, where the two inputs are tied together, and the output is always 0. Any interpretation of the variables ϕ for which $\phi(a) \neq \phi(b)$ will cause the antecedent trajectory to fail, because inputs A and B are electrically equivalent. The only interpretations for which the trajectory succeeds are ones for which the output should be 0, in which case the consequent is also satisfied.

Any checking based purely on testing implications is prone to this sort of "false positive" error. Problems of this sort have been encountered by people using other systems for hardware verification such as HOL [14] and EMC [10]. We believe that shortcomings of this sort can be corrected by more careful attention to the cases where an implication succeeds due a failure of its antecedent. In all of the verification examples we have performed, we make sure that the specification is formulated in such a way that the antecedent trajectory should never fail. We check this as part of the verification process.

9. Conclusions

In terms of mathematical sophistication, the problem solved by our algorithm is far less ambitious than what is attempted by full-fledged temporal logic model checkers. However, we believe that our language is rich enough to be able to describe many important properties of a circuit and to provide a direct path by which such properties may be automatically verified. By keeping the goals of our verifier simple, we obtain an algorithm that is capable of dealing with much larger circuits.

One interesting property of our algorithm, in fact, is that its computational complexity is relatively insensitive to the circuit size. That is, the complexity is determined largely by the complexity of the assertion to be verified, measured in terms of the number of symbolic variables, and the depth of nesting of next time operators. We have found that in many circuits, properties can be expressed in terms of a surprisingly small number of variables. For example, our formulas providing a complete specification of of a k-bit static RAM involve only $2 + 2\log k$ variables. Thus, we can perform the verification in polynomial time irrespective of the heuristic efficiency of the Boolean manipulator.

References

[1] D. L. Beatty, R. E. Bryant, and C.-J. H. Seger, "Synchronous Circuit Verification by Symbolic Simulation: An Illustration," *Sixth MIT Conference on Advanced Research in VLSI*, 1990.

[2] S. Bose, and A. L. Fisher, "Verifying Pipelined Hardware Using Symbolic Logic Simulation," *International Conference on Computer Design*, IEEE, 1989.

[3] S. Bose, and A. L. Fisher, "Automatic Verification of Synchronous Circuits using Symbolic Logic Simulation and Temporal Logic," *IMEC-IFIP International Workshop on Applied*

Formal Methods for Correct VLSI Design, 1989, pp. 759–764.

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

[5] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: a Compiled Simulator for MOS Circuits," *24th Design Automation Conference*, 1987, 9–16.

[6] R. E. Bryant, "Boolean Analysis of MOS Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-6, No. 4 (July, 1987), 634–649.

[7] R. E. Bryant, "Formal Verification of Memory Circuits by Switch-Level Simulation," To appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1990.

[8] J. A. Brzozowski, and M. Yoeli. "On a Ternary Model of Gate Networks." *IEEE Transactions on Computers C-28*, 3 (March 1979), 178–183.

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

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages*, Vol. 8, No. 2 (April, 1986), pp. 244–263.

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

[12] J. A. Darringer, "The Application of Program Verification Techniques to Hardware Verification," *16th Design Automation Conference*, 1979, 375–381.

[13] S. Devadas, H.-K. T. Ma, and A. R. Newton, "On the Verification of Sequential Machines at Differing Levels of Abstraction," *24th Design Automation Conference*, 1987, 271–276.

[14] M. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," *Formal Aspects of VLSI Design*, G. Milne and P. A. Subrahmanyam, *eds.*, North-Holland, 1986, pp. 153–177.

[15] J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg, "A Three-Level Design Verification System," *IBM Systems Journal* Vol. 8, No. 3 (1969), 178–188.

[16] C. A. Mead, and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[17] A. Pnueli, "The Temporal Logic of Programs," *18th Symposium on the Foundations of Computer Science*, IEEE, 1977, pp. 46–56.

[18] D. S. Reeves, and M. J. Irwin, "Fast Methods for Switch-Level Verification of MOS Circuits", *IEEE Transactions on CAD/IC*, Vol. CAD-6, No. 5 (Sept., 1987), pp. 766–779.

[19] C-J. Seger, and R. E. Bryant, “Modeling of Circuit Delays in Symbolic Simulation”, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989, pp. 625–639.