Digital Circuit Verification using Partially-Ordered State Models*

Randal E. Bryant
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA

Carl-Johan H. Seger
Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z4 Canada

Abstract

Many aspects of digital circuit operation can be efficiently verified by simulating circuit operation over "weakened" state values. This technique has long been practiced with logic simulators, using the value *X* to indicate a signal that could be either 0 or 1. This concept can be formally extended to a wider class of circuit models and signal values, yielding lattice-structured state domains. For more precise modeling of circuit operation, these values can be encoded in binary and hence represented symbolically as Ordered Binary Decision Diagrams. The net result is a tool for formal verification that can apply a hybrid of symbolic and partially-ordered evaluation.

1 Introduction

Logic simulators have long employed multiple-valued logic, typically augmenting the binary values 0 and 1 with a value *X* indicating a undefined or unknown signal [11]. This value can be used to indicate an uninitialized state variable or to represent a signal in transition for hazard detection [9]. In earlier work we have shown that such ternary modeling can also be effective when formally verifying circuit correctness [7]. Assuming a monotonicity property of the simulation algorithm, one can ensure that any binary values resulting when simulating patterns containing *X*'s would also result when the *X*'s are replaced by any combination of 0's and 1's. Thus, the number of patterns that must be simulated to verify a circuit can be reduced, often dramatically, by representing many different operating conditions with patterns containing *X*'s. 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*. Using this technique, we have successfully verified random access memory circuit of up to 4096 bits using a conventional switch-level simulator [6]. Considering that such a circuit has over 10^{1233} states, this example illustratives the potential effectiveness of ternary modeling.

*This research was supported by the Defense Advanced Research Projects Agency, ARPA Order Number 4976, by the National Science Foundation, under grant number MIP-8913667, by operating grant OGPO 109688 from the Natural Sciences Research Council of Canada, and by a fellowship from the Advanced Systems Institute.

Ternary modeling is a special case of a more general abstraction technique based on partially-ordered system models. That is, the actual state space of the circuit (in this case all possible combinations of binary values) is extended with values representing sets of circuit states, such that the resulting state set is partially ordered. Lower elements in this ordering are "weaker" in their information content than the actual circuit states. We can therefore verify circuit behavior for a range of different operating conditions by simulating operation on weakened values. By suitable restrictions of the specification notation and the extended next-state function, we can guarantee that any property verified on this more abstract form of simulation must also hold for the original circuit.

Although ternary modeling, or its generalization, allows us to cover many conditions with a single simulation sequence, it lacks the analytic power required for general verification. 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). We therefore also need a method for covering a wide range of operating conditions explicitly, such as symbolic simulation. We have shown that by combining ternary modeling with symbolic simulation [1], we can model even complex sets of behaviors with a single simulation run. With ternary symbolic simulation, the 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. By representing the multiple-valued signals with a binary encoding, we can represent and manipulate them symbolically using Ordered Binary Decision Diagrams (OBDDs) [3].

In this paper we generalize our previous results on ternary simulation to a wider class of partially-ordered system models. It also allows us to apply our methods to more abstract data domains than simple binary-valued signals. We describe a technique for formal verfication, called *symbolic trajectory evaluation* that employs symbolic simulation over partially-ordered values to verify properties of a digital system [5, 8]. This paper focusses on the role

Symbolic?	Model	Patterns	Variables
No	Binary	2^n	0
Yes	Binary	1	n
No	Ternary	$n+1$	0
Yes	Ternary	1	$\lceil \log(n+1) \rceil$

Table 1: Requirements for Verifying n-input AND gate.

	Scalar Cases								Symbolic	
Signal	0	1	2	3	4	5	6	7	High	Low
in0	0	*X*	*X*	*X*	*X*	*X*	*X*	1	$i_2\, i_1\, i_0$	$\overline{i}_2\, \overline{i}_1\, \overline{i}_0$
in1	*X*	0	*X*	*X*	*X*	*X*	*X*	1	$i_2\, i_1\, i_0$	$\overline{i}_2\, \overline{i}_1\, i_0$
in2	*X*	*X*	0	*X*	*X*	*X*	*X*	1	$i_2\, i_1\, i_0$	$\overline{i}_2\, i_1\, \overline{i}_0$
in3	*X*	*X*	*X*	0	*X*	*X*	*X*	1	$i_2\, i_1\, i_0$	$\overline{i}_2\, i_1\, i_0$
in4	*X*	*X*	*X*	*X*	0	*X*	*X*	1	$i_2\, i_1\, i_0$	$i_2\, \overline{i}_1\, \overline{i}_0$
in5	*X*	*X*	*X*	*X*	*X*	0	*X*	1	$i_2\, i_1\, i_0$	$i_2\, \overline{i}_1\, i_0$
in6	*X*	*X*	*X*	*X*	*X*	*X*	0	1	$i_2\, i_1\, i_0$	$i_2\, i_1\, \overline{i}_0$
out	0	0	0	0	0	0	0	1	$i_2\, i_1\, i_0$	$\overline{i}_2 + \overline{i}_1 + \overline{i}_0$

Table 2: Verification of 7-input AND Gate by Ternary Modeling

of partially-ordered states in our methodology, and how these are combined with symbolic simulation to give a trade-off between precision and efficiency. We also show that the formulation in terms of a lattice relations and operations concisely describes the operation of the verifier. Having this mathematical model makes it possible to formally reason about the properties of our tool. A more complete treatment can be found in [12].

Lattice-structured domains based on an information ordering have long been used for giving denotational semantics to programs [13]. In this case the lattice structure is a mathematical device for giving a functional interpretation to a program loop, based on the least fixed point of the function representing one execution of the loop body. Our use of lattice-structured domains is more than a mathematical device—it forms the basis of a tool for automatic hardware verification. We run simulations in which only some of the state components are constituted as full binary values, while the remaining components consist of *X*'s.

2 Illustrative Example

Consider the task of using a simulator to verify that a given circuit has the functionality of an n-input AND gate. Four approaches are tabulated in Table 1, according to whether the simulation is conventional or symbolic, and whether it uses a binary or a ternary system model. With binary modeling, we would need to simulate either 2^n conventional patterns, or a single symbolic pattern of n variables—one per input. In either case, we must, in effect, exhaustively evaluate the circuit functionality.

With ternary modeling, we can exploit the property that if at least one of the inputs to the gate is 0, the output should be 0 regardless of the other inputs. Even with a conventional simulator, we can verify the circuit with just $n+1$ patterns. These are illustrated for the case of $n = 7$ in Table 2. First, there are n patterns that set one input to 0, the remaining to *X*, and checks

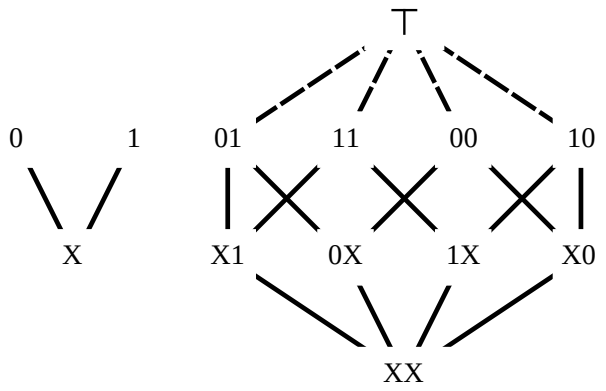

Figure 1: Construction of State Domain from Multiple, Partially-Ordered Signals

that the output is 0. The remaining pattern sets all inputs to 1 and checks that the output is 1.

By the method of *symbolic indexing*, our ternary symbolic simulator can encode all of these cases with a single symbolic pattern [1]. That is, we think of the patterns as being indexed from 0 to n. These index values are then encoded in binary and represented symbolically by a set of $\lceil \log(n+1) \rceil$ index variables. In our example with $n = 7$, we require three index variables: i_2, i_1, and i_0. The signal values are then functions over these index variables mapping to the set $\{0, 1, X\}$. We can in turn represent each of these functions as a pair of Boolean functions, indicating the cases where the signal is 1 ("High") or 0 ("Low"), with the signal otherwise being *X*. Table 2 shows the encoding of the eight ternary patterns by symbolic indexing. The High function is satisfied only when all index variables are assigned value 1, corresponding to the binary representation of 7. The Low function for each signal is satisfied when the index value matches the input number. Thus, each decoding of the index variables corresponds to one of the scalar ternary patterns.

This simple example illustrates how multi-valued modeling can be combined with symbolic simulation. By this method, we can efficiently cover a wide range of circuit operating conditions with a single symbolic simulation pattern involving far fewer variables than would be required for a complete binary symbolic simulation. In the case of an AND gate, we have reduced the number of variables to be logarithmic in the circuit size. For large systems involving many state variables, such reductions can lead to a dramatic improvement in symbolic manipulation efficiency.

Note also that even though we model circuit operation over multiple-valued signals, we utilize binary encodings of these signals so that they can be represented symbolically with OBDDs. This avoids the need to implement special data structures and manipulation algorithms for multi-valued functions. In general, we think of the Boolean variables of the symbolic simulator as providing a set of index variables. Each decoding of the variables covers one of the cases required for verification.

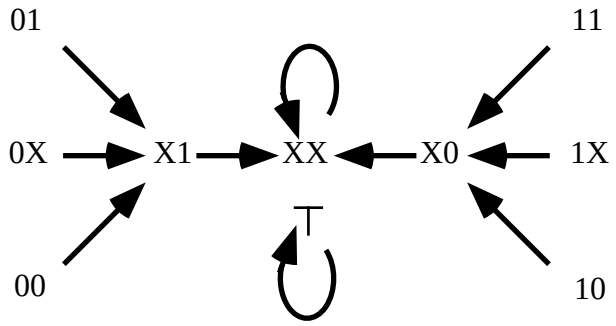

Figure 2: Successor Function for Unit-Delay Inverter (in, out)

3 Mathematical Model

We represent a circuit as a *model structure*, denoted by a tuple $\mathcal{M} = [\langle \mathcal{S}, \sqsubseteq \rangle, Y]$, where $\langle \mathcal{S}, \sqsubseteq \rangle$ is a complete lattice and Y is a monotone successor function $Y: \mathcal{S} \rightarrow \mathcal{S}$. In this model, the *state domain* $\mathcal{S}$ consists of the set of possible circuit states, augmented with additional "weakened" states, plus an additonal "overconstrained" state. These states are partially ordered by a relation $\sqsubseteq$, according to their "information content." The successor function Y represents the excitation function for the circuit, extended to include the additional states.

As an example of a state domain, consider an inverter with input in and output out. We consider the combination of input, output, and internal state to form the overall state of the circuit, and hence there are four possible circuit states: 00, 01, 10, and 11. We then augment the binary signal values to include value *X*, giving five more states *XX*, 0*X*, 1*X*, *X*0, and *X*1. Finally, we add a state $\top$ representing an overconstrained state in the verification. The ordering of these ten states is shown in Figure 1. As indicated on the left side, we consider the individual signals to be ordered by "information content" with *X* less than either 0 or 1. States consisting of multiple signals are ordered according to an elementwise extension of the signal ordering. Thus, we have *XX* as the weakest state, ones contain a single *X* next, and ones consisting of binary values next. As indicated by the dashed lines, the state $\top$ is added as the maximum value in the domain, above any state that corresponds to a set of actual signal values. This example illustrates our general technique for constructing state domains. We start with partially-ordered signal values for each node, form a semi-lattice consisting of vectors of signals ordered element-wise, and complete the lattice with an overconstrained signal $\top$.

Figure 2 illustrates the successor function for a unit-delay inverter, where each state is listed (in, out), and each arc maps a state to its successor. For a unit delay timing model, the successor for a state involves setting each signal to its excitation. In our model, primary inputs, such as in, have excitation *X*, indicating that the circuit itself imposes no constraint on the value of a primary input. Signal out has the (ternary) complement of in as its excitation. Finally, state $\top$ has itself as successor. It can be seen that this successor function is monotone: given states a and b ordered $a \sqsubseteq b$, we have that $Y(a) \sqsubseteq Y(b)$. In other words, strengthening the circuit state can only strengthen the successor.

As this example illustrates, the successor function follows from the circuit representation. Given a logic gate network, we define the excitation for each gate output to be the ternary extension of the logic gate function applied to the gate input signals. The excitation function of a primary input is always *X*. The successor function for the circuit consists of the vector of signal excitations, or is defined to yield $\top$ when the circuit state is $\top$. The restriction that the successor function be monotone follows as a direct property of the ternary algebra. For transistor-level circuits, we derive the excitation functions for the nodes by performing symbolic, switch-level circuit analysis [4]. For higher level models, we may wish to increase the granularity of timing to either a clock phase or clock cycle level, so that one application of the successor function describes a complete phase or cycle of circuit operation.

We view the behavior of a circuit over time as a *trajectory* consisting of an infinite sequence of states $\sigma = \sigma^0\sigma^1 \ldots \in \mathcal{S}^\omega$, obeying the property that

$$Y(\sigma^i) \sqsubseteq \sigma^{i+1} \text{ for } i \geq 0.$$

That is, each state must be consistent with, and possibly stronger than, the result of applying the successor function to the previous state. This rule corresponds to a view of the successor function as defining the constraints imposed on the state sequence by the circuit operation. Such properties as the initial state and the primary input values are either unconstrained or imposed by external agents.

With symbolic evaluation, we model each element of the sequence as a function over a set of Boolean variables $\mathcal{V}$. As described earlier, these variables are viewed simply as indices encoding a number of different cases simulated simultaneously. For any assignment to the variables $\phi: \mathcal{V} \rightarrow \{0, 1\}$, the evaluation $\sigma(\phi)$ denotes some sequence of circuit states. Typically, these coded forms are derived directly from the form of the specification. Similar encoding techniques have been used by Jain and Gopalakrishnan [10] for generating symbolic patterns to cover many binary-valued cases simultaneously.

To implement ternary symbolic simulation, we encode the state of every signal by a pair of OBDDs, similar to the "High" and "Low" encoding used in Table 2. Every ternary logic gate function (or transistor network operation) is expressed as a set of Boolean operations on these encoded values. These operations are in turn implemented using the APPLY operation on the OBDDs [3].

4 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation takes the notion of ternary symbolic simulation one step further by providing a concrete means of specifying the desired behavior of the system operating over time. 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. The relatively simple use of temporal operators is adequate for expressing many of the subtleties of system operation, including clocking conventions and pipelining.

The syntax for symbolic trajectory formulas is defined recursively. It starts as its basis with a set of *simple predicates*. For

circuit models of the form defined earlier, simple predicates are of the form $(n_i$ **is** $s)$, where s is an allowable signal value for node n_i. Such a predicate is true in any state where node n_i has the specified value, as well as in state $\top$. In general, trajectory formulas are constructed as follows:

1. **Simple predicates**: Any simple predicate is a trajectory formula.
2. **Conjunction**: $(F_1 \wedge F_2)$ is a trajectory formula if F_1 and F_2 are trajectory formulas.
3. **Domain restriction**: $(E \rightarrow F)$ is a trajectory formula if F is a trajectory formula and E is a Boolean expression over the set of symbolic Boolean variables $\mathcal{V}$.
4. **Next time**: $(\mathbf{N}F)$ is a trajectory formula if F is a trajectory formula.

A trajectory formula is said to be *instantaneous* when it constains no next-time operators. Such a formula expresses constraints on the system state at only one point in time. Otherwise, a formula expresses contraints for a sequence of states. In particular, the formula $\mathbf{N}F$ states that formula F must hold starting with the successor to the current state. The presence of Boolean expressions makes it possible to specify constraints for multiple operating conditions. For each assignment to the variables in $\mathcal{V}$, the formula can be interpreted to give a different set of constraints.

Due to the restricted syntax of trajectory formulas, we can guarantee that for any formula F, there is a unique minimum state sequence δ_F satisfying this formula. That is, a sequence satisfies the formula if and only if it is greater or equal to (according to an extension of $\sqsubseteq$ to symbolic sequences) sequence δ_F. Had we included either negation or disjunction in the syntax, this property would not hold. The generation of the minimum satisfying sequence is defined recursively according to the structure of the formula. Assume that $\perp = \langle X, X, \ldots, X\rangle$ represents the minimum element of $\mathcal{S}$. Then, state $\overline{p} = \langle X, X, \ldots, X, s, X, \ldots, X\rangle$ is the minimum state satisfying simple predicate $p = (n_i$ **is** $s)$, where the s is in the ith position in the vector, and hence sequence $p, \perp, \perp, \ldots$ is the minimum state sequence satisfying the predicate. Given minimum sequences δ_{F_1} and δ_{F_2} satisfying formulas F_1 and F_2, the minimum sequence satisfying their conjunction is $\delta_{F_1} \sqcup \delta_{F_2}$, where $\sqcup$ represents the least upper bound operation over the lattice of symbolic infinite state sequences. If we consider Boolean expression E to denote a Boolean function e, then formula $E \rightarrow F$ has minimum satisfying sequence $e\ ?\ \delta_F$, where "choice" operator ? yields $\perp, \perp, \ldots$ when its first argument evaluates to 0, or yields its second argument otherwise. Finally, $\perp, \delta_F^0, \delta_F^1, \delta_F^2, \ldots$ must be the minimum sequence satisfying $\mathbf{N}F$. It can also be shown that if the maximum depth of nesting of next time operators in a formula is k, then every element beyond the kth one in the minimum satisfying sequence must be $\perp$.

Our decision algorithm is based on a generalized symbolic simulation. It tests the validity of an *assertion* of the form $[A \Longrightarrow C]$, where both A and C are trajectory formulas. It determines whether or not every trajectory satisfying A (the "antecedent") must also satisfy C (the "consequent"). It does this by simulating the circuit over the constraints implied by the minimum sequence satisfying the antecedent, and testing whether the resulting symbolic state sequence is greater or equal to the minimum sequence satisfying the consequent. Formally, the simulator computes a sequence τ defined as $\tau^0 = \delta_A^0$, and $\tau^i = \delta_A^i \sqcup Y(\tau^{i-1})$. That is, at each step i we combine the external constraint imposed by the antecedent at step i with the internal constraint imposed by the circuit excitation. It can be shown that sequence τ is the minimum trajectory satisfying the antecedent. The assertion is therefore satisfied as long as $\delta_C \sqsubseteq \tau$. Furthermore, if the nesting depth of temporal operators in C is k, we need only run the verification for k steps: beyond this point the assertion is trivially satisfied.

As an example, consider the unit-delay inverter introduced previously. We would specify the behavior of such a circuit by stating that given some value on the input in the current state, the output should have the complement of this value in the next state:

$$\begin{array}{l}\big[(\texttt{in}\ \textbf{is}\ 0) \Longrightarrow \mathbf{N}(\texttt{out}\ \textbf{is}\ 1)\big] \\ \big[(\texttt{in}\ \textbf{is}\ 1) \Longrightarrow \mathbf{N}(\texttt{out}\ \textbf{is}\ 0)\big]\ .\end{array}$$

These formulas can be combined using a symbolic expression over a variable a:

$$\begin{array}{l}\Big(a \rightarrow (\texttt{in}\ \textbf{is}\ 1) \wedge \overline{a} \rightarrow (\texttt{in}\ \textbf{is}\ 0)\Big) \\ \quad\Longrightarrow \\ \mathbf{N}\Big(\overline{a} \rightarrow (\texttt{out}\ \textbf{is}\ 1) \wedge a \rightarrow (\texttt{in}\ \textbf{is}\ 0)\Big)\end{array}$$

For binary signals, we can introduce the notation $(n_i$ **is** $E)$ as a shorthand for the expression $E \rightarrow (n_i$ **is** $1) \wedge \overline{E} \rightarrow (n_i$ **is** $0)$. With this shorthand, we obtain the assertion:

$$\big[(\texttt{in}\ \textbf{is}\ a) \Longrightarrow \mathbf{N}(\texttt{out}\ \textbf{is}\ \overline{a})\big]\ .$$

For the n-input AND gate verification described in Section 2, we could specify its desired behavior by two assertions. The first would cover all of the cases where at least one input is 0:

$$\big[(\texttt{in}[i]\ \textbf{is}\ 0) \Longrightarrow \mathbf{N}(\texttt{out}\ \textbf{is}\ 0)\big]$$

The second would cover the case where all inputs are 1:

$$\begin{array}{l}(\texttt{in}[0]\ \textbf{is}\ 1) \wedge (\texttt{in}[1]\ \textbf{is}\ 1) \wedge \cdots \wedge (\texttt{in}[n-1]\ \textbf{is}\ 1) \\ \quad\Longrightarrow \\ \mathbf{N}(\texttt{out}\ \textbf{is}\ 1)\end{array}$$

With our current implementation, we would translate these two assertions separately. For $n = 7$ the first assertion would yield symbolic patterns corresponding to cases 0–6 in Table 2. The indexing of the vector of input nodes translates directly into the coding shown for the "Low" column in the table, while the "High" value would be the constant function **0** for all inputs. The second assertion would yield case 7 of this table, with no symbolic values. One could imagine a more sophisticated translator that could pack all eight cases into the single set of symbolic patterns shown in the table.

An important property of our algorithm is that it requires a comparatively small amount of simulation and symbolic manipulation to verify an assertion. Due to the restrictions of the formula syntax, we can verify an assertion by a single symbolic simulation involving only variables explicitly mentioned in the assertion.

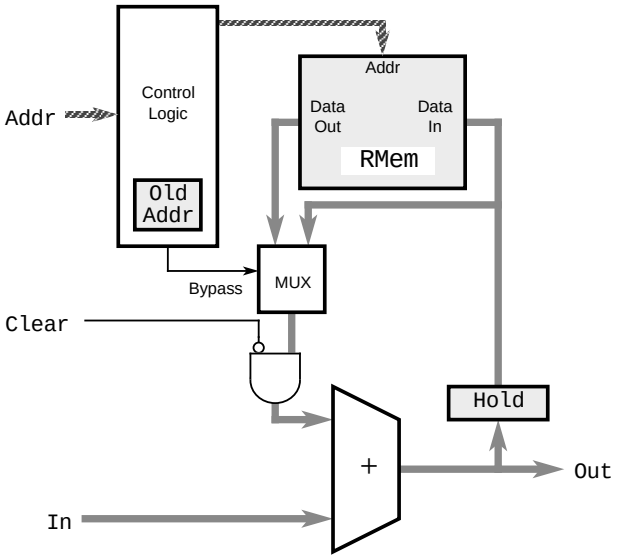

Figure 3: Pipelined Addressable Accumulator Circuit

The length of the simulation sequence depends only on the depth of nesting of temporal next-time operators in the assertion.

5 Example Applications

As an example, consider a pipelined implementation of an "addressable accumulator" circuit, illustrated in Figure 3. This circuit maintains the sums for m different channels. On each cycle, the address input `Addr` specifies which sum to update, the clear signal `Clear` indicates whether or not to reset the sum, and the data input `In` gives the data value to accumulate. For efficiency, the write-back to the register file is overlapped with the operation of the adder. Bypass logic is included to handle the case where the same address is given in two successive cycles. In this case, the control logic detects that the current address matches the saved previous address (stored in register `OldAddr`, and transfers the data from the ALU output register `Hold` directly into the adder.

In specifying the desired behavior of the circuit, we split the specification into two parts: an *abstract model* expressing the high level behavior without considering the pipeline structure, and a *state mapping* describing how the state of the abstract model is realized by the circuit. This mapping includes sufficient information about the pipeline structure to perform the verification. The actual symbolic assertions are derived by combining these two parts of the specification [2, 8].

At the abstract level, we want to express three properties of the addressable accumulator. These can be expressed as three assertions, each over vectors of Boolean variables representing possible address ($\vec{u}$, $\vec{v}$) and data ($\vec{a}$, $\vec{b}$) values. At this level, we define the state of the system in terms of an abstract register array. That is, we introduce predicate $\mathrm{Reg}[\vec{u}, \vec{a}]$ stating that the accumulated sum with address $\vec{u}$ has value $\vec{a}$. In each assertion, we provide just enough context in the antecedent to be able to describe some aspect of the next state. The full specification consists of the conjunction of all of the assertions for all valuations of the symbolic variables.

1. With `Clear` set to 1, the addressed value should be set to the input data:

$$(\texttt{Addr}\ \textbf{is}\ \vec{u}) \wedge (\texttt{Clear}\ \textbf{is}\ 1) \wedge (\texttt{In}\ \textbf{is}\ \vec{a}) \implies \mathbf{N}\left(\mathrm{Reg}[\vec{u}, \vec{a}]\right)$$

2. With `Clear` set to 0, the addressed value should be incremented by the input data.

$$(\texttt{Addr}\ \textbf{is}\ \vec{u}) \wedge (\texttt{Clear}\ \textbf{is}\ 0) \wedge \mathrm{Reg}[\vec{u}, \vec{b}] \wedge (\texttt{In}\ \textbf{is}\ \vec{a}) \implies \mathbf{N}\left(\mathrm{Reg}[\vec{u}, sum(\vec{a}, \vec{b})]\right)$$

   where *sum* represents the vector of Boolean functions describing binary addition.

3. Any unaddressed location should remain unchanged:

$$(\texttt{Addr}\ \textbf{is}\ \vec{u}) \wedge \mathrm{Reg}[\vec{v}, \vec{b}] \implies \vec{u} \neq \vec{v} \rightarrow \mathbf{N}\left(\mathrm{Reg}[\vec{v}, \vec{b}]\right)$$

The state mapping describes how this abstract register array is realized by the circuit, i.e., where the accumulated values are stored. Normally, sums are stored in the actual register memory, `RMem`. Conditionally, a sum can be in the pipeline register `Hold`. This is expressed by defining the predicate $\mathrm{Reg}[\vec{u}, \vec{a}]$ as:

$$\exists \vec{w} \left[\begin{array}{lll} \texttt{OldAddr}\ \textbf{is}\ \vec{w} & \wedge & (\vec{u} = \vec{w} \rightarrow \texttt{Hold}\ \textbf{is}\ \vec{a}) \\ & \wedge & (\vec{u} \neq \vec{w} \rightarrow \texttt{RMem}[\vec{u}]\ \textbf{is}\ \vec{a}) \end{array} \right]$$

In this mapping, we use a vector of existentially quantified variables $\vec{w}$ to indicate that register `OldAddr` may hold an arbitrary address independent of the current operation. Given such an address, abstract register $\vec{u}$ maps to either the holding register ($\vec{u} = \vec{w}$), or to register $\vec{u}$ in the register file ($\vec{u} \neq \vec{w}$). Adding quantified variables to our assertion logic is straightforward.

To verify the circuit, the high level specification assertions are combined with the state mapping, and additional information about circuit clocking and timing is included. The result is a set of symbolic assertions that can be verified by our switch-level symbolic trajectory evaluator. We have used this to verify accumulator designs with up to 32 registers of 32 bits each, requiring less than 10 minutes of CPU time on a workstation. Observe that even though we are verifying systems with over 1000 state variables, our symbolic evaluation requires less than 80 Boolean variables. The resulting patterns make extensive use of symbolic indexing, where the variables comprising symbolic addresses $\vec{u}$ and $\vec{v}$ index which of the registers hold actual data values, and which have value *X*. This demonstrates the efficiency gains of a partially-ordered state model.

For more complex systems, the detailed modeling in terms of individual circuit elements becomes impractical. In complex pipelines, the control logic is typically more problematic than the data path design. We would therefore like to abstract the details of

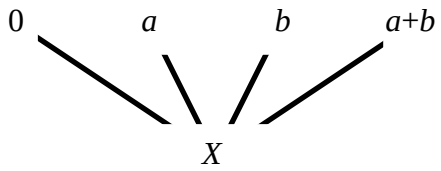

Figure 4: Multi-Valued Data Domain for Pipeline Verification

+	a	b	$a+b$	0	X
a	X	$a+b$	X	a	X
b	$a+b$	X	X	b	X
$a+b$	X	X	X	$a+b$	X
0	a	b	$a+b$	0	X
X	X	X	X	X	X

Table 3: Definition of Addition for Multi-Valued Data Domain

the data path and instead concentrate on the data transfers caused by the control logic. For this task we can exploit the generality of our partially-ordered state model. We can construct a special multi-valued signal domain to represent "interesting" values for the data path, as illustrated in Figure 4. This domain includes elements a and b to represent arbitrary data values, the value 0 to enable modeling the effect of the clear signal, and a special element $a+b$ to indicate the sum of values a and b. Any other value is simply classified as an unknown value *X*. We define the operation of the adder over this data domain according to Table 3. Observe how this table assigns value *X*, except for those cases where the adder value can be clearly categorized as one of the other values.

To verify the pipelined accumulator circuit by this method, we would systematically simplify the data path hardware description to one operating on abstract data values, while maintaining the detailed model of the control logic. The assertions to be verified would be similar to the earlier ones, except that references to vectors of binary data $\vec{a}$, $\vec{b}$, and $sum(\vec{a},\vec{b})$ would be changed to data elements a, b, and $a+b$, respectively. The resulting verification would require just 15 variables for a 32 register accumulator, and would also be independent of the word size. Thus, our generalization from ternary logic to partially-ordered state domains enables us to exploit forms of data abstraction in verifying complex hardware.

6 Conclusions

We have shown that partially-ordered states, a generalization of ternary modeling, can be combined with symbolic evaluation to provide a powerful tool for formal circuit verification. By defining the behavior of the symbolic verifier in terms of lattice operations, we can clearly state the functionality of our tool and prove that it does indeed formally verify temporal properties of the circuit.

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] D. L. Beatty, "A Methodology for Formal Verification, with Application to Microprocessors," PhD Thesis, Carnegie Mellon University, 1993. Available as Technical Report CMU-CS-93-190.

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

[4] 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.

[5] R. E. Bryant, and C.-J. H. Seger, "Formal Verification of Digital Circuits Using Symbolic Ternary System Models," *Computer-Aided Verification '90*, E. M. Clarke, and R. P. Kurshan, *eds*. American Mathematical Society, 1991, pp. 121–146.

[6] R. E. Bryant, "Formal Verification of Memory Circuits by Switch-Level Simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 10, No. 1 (January, 1991), pp. 94–102.

[7] R. E. Bryant, "A Methodology for Hardware Verification Based on Logic Simulation," *J.ACM*, Vol. 38, No. 2 (April, 1991), pp. 299–328.

[8] R. E. Bryant, D. E. Beatty, and C.-J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," *28th Design Automation Conference*, June, 1991.

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

[10] P. Jain, and G. Gopalakrishnan, "Hierarchical Constraint Solving in the Parameteric Form with Applications to Efficient Symbolic Simulation based Verification," *International Conference on Computer Design*, IEEE, 1993, pp. 304–307.

[11] 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.

[12] C.-J. Seger, and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," submitted for publication, April, 1993.

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