7-1995

Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories

Carl-Johan H. Seger
University of British Columbia

Randal E. Bryant
Carnegie Mellon University

Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories*

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

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

July 1, 1999

Abstract

Symbolic trajectory evaluation provides a means to formally verify properties of a sequential system by a modified form of symbolic simulation. The desired system properties are expressed in a notation combining Boolean expressions and the temporal logic "next-time" operator. In its simplest form, each property is expressed as an assertion $[A \Longrightarrow C]$, where the antecedent A expresses some assumed conditions on the system state over a bounded time period, and the consequent C expresses conditions that should result. A generalization allows simple invariants to be established and proven automatically.

The verifier operates on system models in which the state space is ordered by "information content". By suitable restrictions to the specification notation, we guarantee that for every trajectory formula, there is a unique weakest state trajectory that satisfies it. Therefore, we can verify an assertion $[A \Longrightarrow C]$ by simulating the system over the weakest trajectory for A and testing adherence to C. Also, establishing invariants correspond to simple fixed point calculations.

This paper presents the general theory underlying symbolic trajectory evaluation. It also illustrates the application of the theory to the task of verifying switch-level circuits as well as more abstract implementations.

1 Introduction

Verifying a digital system by conventional simulation is feasible only for very small systems, since the large number of possible initial states and input sequences would require massive amounts of case analysis. By exploiting a combination of *abstraction* and *symbolic manipulation*, on the other hand, symbolic trajectory evaluation can verify the behavior of complex systems by a modified form of simulation. This method exploits abstraction by extending the system state space to include elements representing sets of actual states, yielding a partially-ordered system model. A single simulation sequence can then verify that the system would produce a unique result for a set of

*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 and Engineering Research Council of Canada, and by a fellowship from the B. C. Advanced Systems Institute.

initial states or input sequences. It exploits symbolic manipulation by a modified form of symbolic simulation. The Boolean expressions appearing in the system specification are converted into symbolic patterns for the simulator. Like a conventional simulation, a single run of the trajectory evaluator models the system behavior over a single state sequence, although this sequence is both symbolic and partially-ordered.

1.1 Partially-Ordered System Modeling

In earlier work, we demonstrated the utility of ternary modeling for verifying a variety of circuits [11, 12]. Our methodology was based on ternary simulation of VLSI circuits, where a third value X is added to the set $\{0, 1\}$ of possible signal values, indicating an unknown or indeterminate logic value. Assuming a monotonicity property of the simulation algorithm, one can ensure that any binary (i.e., 0 or 1) 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 often be reduced dramatically by representing many different operating conditions by 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. This requires far less effort than analyzing the effect of the action on all possible initial binary states.

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. With ternary simulation, a state with some nodes set to X covers those circuit states obtained by replacing the X values with all combinations of 0 and 1. The state with all nodes set to X thus covers all possible actual circuit states. By extending the next-state function of the circuit to one over the expanded state set, we can verify circuit behavior for a set of different operating conditions with a single simulation run. By suitable restrictions of the specification syntax 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.

In this paper we generalize our previous results on ternary simulation to a wider class of partially-ordered system models. This generalization simplifies the presentation by allowing us to focus on the essential properties of the abstraction technique while eliminating artifacts specific to ternary modeling. It also allows us to apply our methods to higher level data domains than simple binary-valued signals.

1.2 Symbolic Simulation

Although ternary modeling, or its generalization, allows us to cover many conditions with a single simulation sequence, it lacks the analytic power required for complete verification, except for restricted classes of circuits such as memories [11]. We have shown that by combining ternary modeling with symbolic simulation [1], we can model even more 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. In essence, ternary symbolic simulation allows us to combine multiple ternary simulation sequences into a single symbolic sequence.

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 $X + \overline{X} = X$, where $+$ and $\overline{\ }$ 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.

1.3 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation takes the notion of ternary symbolic simulation one step further by providing a concrete means of specifying and verifying the desired behavior of the system operating over time. In earlier papers [9, 13], we introduced the notion of symbolic trajectory evaluation for ternary system models and demonstrated its utility on several actual circuits. In this paper we generalize the technique to a wider class of system models and specifications. We also make our previous, informal claims more precise and rigorous.

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. 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 [16, 32]. Nonetheless, we have found that we can readily express many aspects of synchronous digital systems at various levels of abstraction. It is quite adequate for expressing many of the subtleties of system operation, including clocking conventions and pipelining.

Our decision algorithm is based on a generalized symbolic simulation. In its simplest form 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.

A more complex condition of the form $[A \Longrightarrow C]^*; G$ can also be verified, where A and C are trajectory formulas and G is an assertion. Intuitively, the formula is deemed to hold if and only if for every sequence of states the system may go through, if the state sequence satisfies some number of iterations of A, then it must also satisfy the same number of iterations of C and furthermore the remaining sequence must satisfy G. Assertions of this form are useful for verifying circuits that may remain in an idle state for an unbounded amount of time, e.g., for a processor held in a "wait-state" by the memory subsystem. Our verification method proves invariants of this form by using symbolic simulation to compute a fixed-point which intuitively serves as a "summary" of what states the system can be in after it has gone though any number of iterations of A.

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 [5], 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 and the speed of convergence of the fixed-point calculations.

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

1.4 Illustrative Example

To illustrate the combined use of partially-ordered system modeling and symbolic simulation, 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 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.

1.5 Related Work

Our approach to verification relates most closely to the symbolic model checking algorithms devised by a number of researchers [5, 15, 19]. Like our program, these algorithms verify that a finite state system, modeled symbolically, obeys a property expressed in temporal logic. Despite these general 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 systems that can be modeled and in the properties that can be verified. For example, other symbolic model checkers [15] can model an arbitrary, nondeterministic system, since the system is described by a transition relation. The symbolic verifier for LUSTRE programs [25] is based on a next-state function representation of system behavior, but allows the inputs on each time step to be chosen nondeterministically. In computing circuit behavior by a form of simulation, our method effectively models system behavior in terms of the next-state functions for the circuit state elements. Furthermore, we do not choose inputs nondeterminstically, but rather constrain the inputs according to the antecedents of the assertion formulas. We can model some forms of nondeterministic behavior by encoding a set of possible states with the value corresponding to the greatest lower bound in the partial ordering. This form of modeling would yield overly pessimistic results for highly divergent system behaviors.

Although our method does not compare favorably to symbolic model checking for verifying highly nondeterministic systems, we can efficiently model circuit behavior with more detailed circuit and timing models. In particular, we can handle most of the techniques found in (discrete) circuit simulators, including switch-level models, arbitrary clocking schemes, and various delay models for the circuit elements. Our verifier is thus one of the few that can model system behavior at a level of timing granularity finer than complete clock cycles.

In its most general form, verification by symbolic model checking can decide a class of formulas consisting of a complete branching time, propositional temporal logic. Our method can only be used to verify properties of bounded state sequences, intermixed with periods of invariant behavior. Our restrictions on the formulas to be checked allow us to verify system behavior by simulating circuit behavior over a single, symbolic state sequence. There are other precedents for restricting the class of formulas that can be checked in order to improve the efficiency of the verifier. For example, Clarke, Grumberg, and Long [17] have shown that by restricting the use of negation and existential operators, they can reliably replace a detailed system model by a more simplified abstraction. Any property proved of the abstract model is guaranteed to hold for the more detailed one. Similarly, we prohibit negation and even disjunction in our logic to make it possible to conservatively approximate the circuit behavior by a single, symbolic state sequence.

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.

Most other automated approaches to sequential circuit verification are based on testing state machine equivalence [18, 21]. 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.

Symbolic simulation has been proposed by others as a hardware verification technique. Bose and Fisher have shown that these methods can be applied to complex circuits, including ones with pipelining [4]. Their method, however, requires a complete characterization of the system by binary symbolic simulation. That is, the user identifies each place state is stored in the circuit, either as charge on a node, or as a pair of complementary values within a static memory element. They then symbolically simulate a single clock cycle, where each state variable and each input signal is represented by a distinct Boolean variable, yielding a complete characterization of the next-state functions for every state variable. 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.

Other researchers have suggested symbolic simulation as a means of circuit verification [20, 33]. None of this work has presented a clear methodology for sequential circuit verification, however.

1.6 Outline of Paper

This paper presents the theoretical basis for symbolic trajectory evaluation. Following a summary of the mathematical foundations, we describe the concept of partially-ordered system models and how a system can be represented by the language consisting of all possible compatible state sequences, referred to as trajectories. Next we introduce a "scalar" version of the specification notation, where only constant expressions are permitted. We show that any assertion in this notation can be verified by simulating the (unique) weakest state sequence satisfying the antecedent and testing adherence to the consequent. We then show that the concepts generalize to the symbolic case, where the specifications may contain expressions over a set of Boolean variables. One can view a symbolic assertion as simply encoding a number of scalar assertions that can then be evaluated simultaneously through symbolic simulation. Finally, we discuss some of the practical issues of implementing and applying our theory to real-life digital circuits.

2 Mathematical Background

In this section we give concise definitions of many concepts used throughout the paper. Readers unfamiliar with the notation of lattice theory may wish to refer an introductory text for additional information.

In general, we use calligraphic letters $\mathcal{A}, \mathcal{B}, \ldots$, to denote sets and lower case letters, $a, b, \ldots$, to denote individual elements of sets. Unless otherwise stated, all sets are assumed to be finite.

The *cartesian product* $\mathcal{A} \times \mathcal{B}$ of two sets $\mathcal{A}$ and $\mathcal{B}$ is the set of all ordered pairs (a, b), where $a \in \mathcal{A}$ and $b \in \mathcal{B}$. A *binary relation on* a set $\mathcal{B}$ is any subset of $\mathcal{B} \times \mathcal{B}$. Let R be a binary relation on $\mathcal{B}$, i.e., $R \subseteq \mathcal{B} \times \mathcal{B}$. We say that R is *reflexive* iff aRa for all $a \in \mathcal{B}$. Similarly, R is *antisymmetric* iff aRb and bRa implies $a = b$ for all $a, b \in \mathcal{B}$. Finally, R is *transitive* iff aRb and bRc implies aRc

for all $a, b, c \in \mathcal{B}$. A binary relation on $\mathcal{B}$ which is reflexive, antisymmetric, and transitive is called a *partial order* on $\mathcal{B}$.

A *poset* (partially ordered set) is an ordered pair $\langle \mathcal{S}, \sqsubseteq \rangle$, where $\mathcal{S}$ is a set and $\sqsubseteq$ is a partial order on $\mathcal{S}$. Intuitively, we will view a partial order as ordering the values by their "information content." That is, elements less than others "contain less information".

If $\langle \mathcal{S}, \sqsubseteq \rangle$ is a poset, $\mathcal{A} \subseteq \mathcal{S}$, and $b \in \mathcal{S}$, then b is a *lower bound* of $\mathcal{A}$ iff $b \sqsubseteq a$ for all $a \in \mathcal{A}$. A lower bound a of $\mathcal{A}$ is called *greatest lower bound* of $\mathcal{A}$, written $glb(\mathcal{A})$, if and only if $b \sqsubseteq a$ for every lower bound b of $\mathcal{A}$. The concept of *upper bound* and *least upper bound* of $\mathcal{A}$, written $lub(\mathcal{A})$, are defined dually. If $\mathcal{A} = \{a, b\}$, we will write $glb(a, b)$ ($lub(a, b)$) rather than $glb(\{a, b\})$ ($lub(\{a, b\})$). Clearly, if $glb(\mathcal{A})$ exists, it is unique, and the same holds for $lub(\mathcal{A})$.

A poset $\langle \mathcal{S}, \sqsubseteq \rangle$ is said to have a *universal lower bound* $\bot \in \mathcal{S}$ iff $\bot \sqsubseteq a$ for every element $a \in \mathcal{S}$. A poset is said to have a *universal upper bound* $\top \in \mathcal{S}$ iff $a \sqsubseteq \top$ for every element $a \in \mathcal{S}$.

A poset $\langle \mathcal{S}, \sqsubseteq \rangle$ is a *complete lattice* if $lub(\mathcal{A})$ and $glb(\mathcal{A})$ exist for every subset $\mathcal{A} \subseteq \mathcal{S}$. Given that $\mathcal{S}$ is a finite set, one can show [37] that if $lub(a, b)$ and $glb(a, b)$ exist for every $a, b \in \mathcal{S}$, then $\langle \mathcal{S}, \sqsubseteq \rangle$ is a complete lattice. Note that, by definition, every complete lattice has a universal upper bound $\top \in \mathcal{S}$ and a universal lower bound $\bot \in \mathcal{S}$.

If $\langle \mathcal{S}_1, \sqsubseteq_1 \rangle, \ldots, \langle \mathcal{S}_n, \sqsubseteq_n \rangle$ are n complete lattices let $\mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$ and for any $a, b \in \mathcal{S}$ let $a \sqsubseteq b$ iff $a_i \sqsubseteq_i b_i$ for $1 \leq i \leq n$. It is easy to verify that $\langle \mathcal{S}, \sqsubseteq \rangle$ forms a complete lattice.

A *mapping* $f: \mathcal{A} \to \mathcal{B}$ consists of a function f assigning an element b from the *codomain* $\mathcal{B}$ to each element a of its *domain* $\mathcal{A}$, written as $b = f(a)$.

Given a poset $\langle \mathcal{S}, \sqsubseteq \rangle$ and a mapping $f: \mathcal{S} \to \mathcal{S}$, we say that f is *monotone* iff

$$a \sqsubseteq b \implies f(a) \sqsubseteq f(b)$$

This monotonicity definition is consistent with our use of information content. If a mapping is monotone, we cannot "gain" any information by reducing the information content of the arguments to the function.

A *predicate* over $\mathcal{S}$ is a special type of mapping $\mathcal{S}$ to the complete lattice with elements *false* and *true*, with *false* as the universal lower bound and *true* as the universal upper bound. A predicate p is said to be *simple* iff p is monotone and there is a unique element $\overline{p} \in \mathcal{S}$, called the *defining value*, such that $p(t) = true$ iff $\overline{p} \sqsubseteq t$ for all $t \in \mathcal{S}$. Another way of stating this property is that p is a simple predicate iff p is monotone and $p(glb(\{s \in \mathcal{S} | p(s) = true\})) = true$.

A fixed-point of a mapping $f: \mathcal{S} \to \mathcal{S}$ is a value a such that $a = f(a)$. Furthermore, if $\langle \mathcal{S}, \sqsubseteq \rangle$ is a complete lattice and f is monotone, then the mapping has a unique *greatest fixed-point*, i.e., a fixed-point a such that $a' \sqsubseteq a$ for any other fixed-point a'. This fixed-point is denoted *Gfp a*. $f(a)$. Furthermore, for the case where $\mathcal{S}$ is finite, this fixed-point can be derived by iteratively computing $a^0 = \top$, and $a^i = f(a^{i-1})$ for $i > 0$. Eventually some iteration step will yield $a^i = a^{i-1}$; this value is the greatest fixed-point [37].

To express the behavior of a system working over time, we will reason about *sequences* of elements from some set $\mathcal{S}$. Conceptually, we will consider the sequences to be infinite, although the properties we will express can always be determined from some bounded length prefix of the sequence. Given a poset $\langle \mathcal{S}, \sqsubseteq \rangle$, we extend the relation $\sqsubseteq$ to sequences pointwise. That is, if $\sigma = \sigma^0 \sigma^1 \ldots$ and $\tau = \tau^0 \tau^1 \ldots$ are two infinite sequences of elements from $\mathcal{S}$, then $\sigma \sqsubseteq \tau$ iff $\sigma^i \sqsubseteq \tau^i$ for $i \geq 0$. Similarly, the definitions of *lub* and *glb* are extended pointwise. Finally, for notational convenience, if $\sigma = \sigma^0 \sigma^1 \sigma^2 \ldots$ we will often write σ as $\sigma^0 \tilde{\sigma}$, where $\tilde{\sigma} = \sigma^1 \sigma^2 \ldots$.

3 Model Structure

The model we use of a system is simple and general. A *model structure* is 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 $Y : \mathcal{S} \rightarrow \mathcal{S}$. Intuitively, the successor function is used to express constraints on the permissible sequences. In other words, given that the system is in state $s \in \mathcal{S}$, we view $Y(s)$ as denoting the least specified state the system can be in one time unit later. Here, "least specified" is defined in terms of the partial order $\sqsubseteq$.

3.1 Structure Example

In order to make the theory easier to follow but also to provide a concrete application for the general theory, we will use switch-level circuit verification as a running example throughout the paper. There are several reasons for this. First, there is a historical reason since this work grew out of switch-level simulation and verification. Secondly, there is a very close connection between our notion of a model structure and the type of models that are used in switch-level simulation. Nonetheless, the underlying concepts apply to more general classes of systems, examples of which will be given later.

In switch-level models it is useful to allow each circuit node to take on one of three distinct values. Let $\mathcal{T} = \{0, 1, X\}$ denote such a set of values. There are several advantages in extending the domain from $\{0, 1\}$ to $\mathcal{T}$. 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 in the course of normal circuit operation. This occurs frequently when modeling circuits at the switch-level [8], 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 a value X to the offending circuit nodes, indicating that the actual digital value cannot be determined [14, 28]. Thus the value X is introduced to denote an "unknown" and possibly indeterminate value.

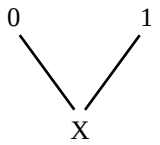

Figure 1: The $\leq$ partial order.

In order to formalize this concept of an "unknown" value, define the partial order $\leq$ on $\mathcal{T}$ as follows: $a \leq a$ for all $a \in \mathcal{T}$, $X \leq 0$, and $X \leq 1$. In Fig. 1 we show the Hasse diagram for the partial order. We can view this partial ordering as ordering values by their "information content." That is, X indicates an absence of information while 0 and 1 represent specific, fully-defined values.

Let $\mathcal{T}^m$, $m \geq 1$, denote the set of all possible vectors of ternary values of length m, i.e., $\{\langle a_1, \ldots, a_m \rangle | a_i \in \mathcal{T}, 1 \leq i \leq m\}$. The partial order $\leq$ is extended to $\mathcal{T}^n$ pointwise: $\vec{a} \leq \vec{b}$ iff $a_i \leq b_i$ for $1 \leq i \leq m$. Unfortunately, $\langle \mathcal{T}^m, \leq \rangle$ is not a complete lattice, since the least upper bound does not exist for every pair of elements in $\mathcal{T}^m$. We solve this by introducing a new top element. In other words, let $\mathcal{C} = \mathcal{T}^m \cup \{\top\}$. Intuitively, one can view $\top$ as representing an "overconstrained" state, i.e., a state vector in which some node is both 0 and 1 at the same time. Let $\sqsubseteq$ be the partial order on $\mathcal{C}$ defined as follows: $s \sqsubseteq \top$ for every $s \in \mathcal{C}$ and if $\vec{s}, \vec{t} \in \mathcal{T}^m$ then

$\vec{s} \sqsubseteq \vec{t}$ iff $\vec{s} \leq \vec{t}$. Clearly, $\langle \mathcal{C}, \sqsubseteq \rangle$ forms a complete lattice in which $\perp = X, \ldots, X$. Thus we now have the first half of a model structure.

The underlying model of a switch-level circuit we use is quite simple, as well as general. A circuit is a tuple $(\mathcal{N}, \vec{y})$, where $\mathcal{N}$ is a set of nodes and $\vec{y}$ is a vector of excitation, or next-state, functions. In the mathematical presentation we will refer to the nodes as $n_1, n_2, \ldots, n_m$, whereas in our examples we often will use more descriptive names.

Since X is meant to denote an unknown value, a gate with an X on its input must treat this value in a conservative way. Consequently, the excitation functions are required to be monotone with respect to the partial order $\leq$. This monotonicity requirement 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 next-state values, or it will change some binary values to X.

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 $\vec{a} \in \mathcal{T}^m$. 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 $y_{n_i}: \mathcal{T}^m \rightarrow \mathcal{T}$ determined by the circuit topology and functionality.

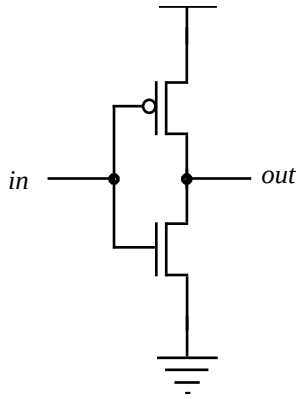

Figure 2: CMOS inverter.

To illustrate our notion of excitation function, consider the CMOS circuit shown in Fig. 2. In Fig. 3 we give a graphical representation of the next-state function assuming the circuit behavior is analyzed using a unit-delay model. Note that no matter what the current state is, the next-state function for the input is X. Also, if the current input is binary, it is easy to see that the output one time unit later will be the complement of this value.

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 [34], both nominal delay and bounded delay circuit models can be used. However, to make our example as simple as possible, we will use

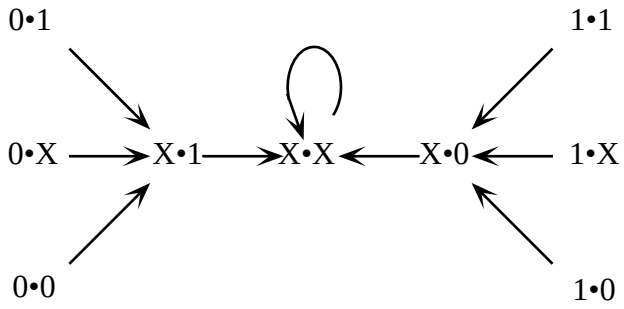

Figure 3: Excitation function of unit delay inverter ($in \cdot out$).

a unit delay model unless otherwise stated.

In order to obtain a model structure, we only need to define a monotone next time function mapping $\mathcal{C}$ to $\mathcal{C}$. We do this by extending $\vec{y}$ from $\mathcal{T}^m \to \mathcal{T}^m$ to $\mathcal{C} \to \mathcal{C}$ in the obvious way. Thus define:

$$Y(a) = \begin{cases} \vec{y}(a) & \text{if } a \in \mathcal{T}^m \\ \top & \text{otherwise} \end{cases}$$

Clearly, Y is monotone and thus $\mathcal{M}^{\mathcal{C}} = [\langle \mathcal{C}, \sqsubseteq \rangle, Y]$ forms a model structure.

3.2 Trajectories

Let us now return to the more general theory in which $[\langle \mathcal{S}, \sqsubseteq \rangle, Y]$ is any model structure. Let $\mathcal{S}^\omega$ denote the set of all infinite sequences of elements from $\mathcal{S}$. In general, sequences are useful when reasoning about model behaviors. However, not all sequences represent possible behaviors of a model. The successor function generally restricts the possible sequences significantly. We formalize this property by introducing the concept of a trajectory. Given a model $\mathcal{M}$ and an arbitrary sequence $\sigma = \sigma^0\sigma^1 \ldots \in \mathcal{S}^\omega$ we say that the sequence is a *trajectory* if and only if

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

This rule for trajectories is consistent with our view of the successor function, i.e., a function computing a constraint on the possible value of the successor state. Another way of describing the next-state function is to view it as computing the most general state the system can evolve into during the next time step given its current state.

The set of all trajectories of model $\mathcal{M}$ is denoted $L(\mathcal{M})$. Occasionally it is convenient to restrict the set of trajectories by requiring the first state in the trajectory to be greater than or equal to some element in $\mathcal{S}$. Consequently, define

$$L(\mathcal{M}, z) = \{\sigma^0\sigma \mid \sigma^0\sigma \in L(\mathcal{M}) \text{ and } z \sqsubseteq \sigma^0\}.$$

Note that $L(\mathcal{M}, \bot) = L(\mathcal{M})$.

The following proposition follows trivially from the definition of trajectories:

Proposition 1 *If $\sigma = \sigma^0\sigma^1\sigma^2 \ldots \in L(\mathcal{M})$ then $\sigma^1\sigma^2 \ldots \in L(\mathcal{M})$. In other words, the set $L(\mathcal{M})$ is suffix-closed, i.e. every suffix of every trajectory in $L(\mathcal{M})$ is also in $L(\mathcal{M})$.*

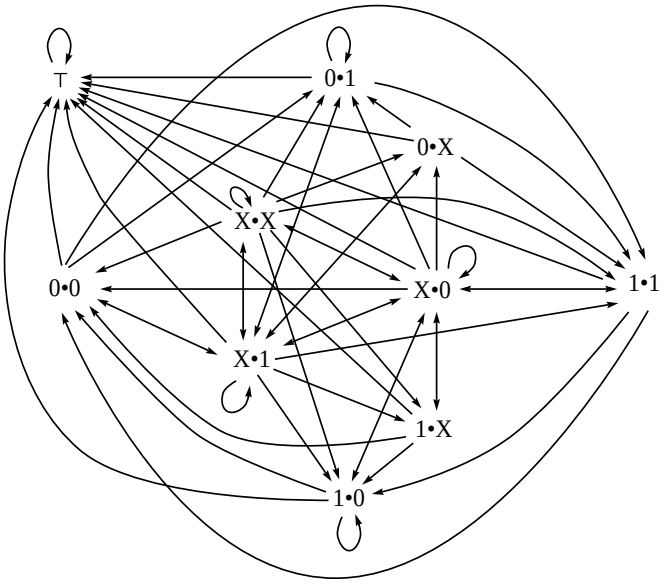

Figure 4: $L(\mathcal{M}^{\mathcal{C}})$ for a unit delay inverter.

Another way of stating Proposition 1 is to say that we assume that every state in $\mathcal{S}$ is a possible initial state of the system.

In Fig. 4 we illustrate the set of all trajectories ($L(\mathcal{M}^{\mathcal{C}})$) for the unit delay inverter described earlier. In this figure, the set of labels encountered while traversing any infinite path in the graph denotes a trajectory. Before discussing this graph further, recall that the $\top$ state is used to represent overconstrained states. In a matter of speaking, we consider that `in` is both 0 and 1 at the same time in the state $\top$. A similar remark holds for `out`. In view of this interpretation, we can draw several conclusions from the graph. For example, we can see that for every trajectory $\sigma^0\sigma^1\ldots$ such that `in` is 1 in σ^0 we have that `out` is 0 in σ^1. The same statement holds with 0 replaced by 1 and 1 replaced by 0. At its core, our verification methodology establishes properties such as these for a given model structure. More specifically, in the next section we define a small logic that allows us to state properties like the ones above in a concise and unambiguous way. We then define an efficient way of determining whether the formulas in the logic are valid for a particular model structure. One main contribution of the paper is the development of a checking algorithm that only needs to explore a tiny fraction of the complete state graph as opposed to how it is shown in Fig. 4.

4 Specification Language

The basic specification language we use is very simple. In fact, at a first glance it might appear as if it can only be used to specify rather trivial behaviors. However, this is a bit of an illusion. In particular, we will later in the paper extend the model structure to a symbolic domain and give several examples of how non-trivial behaviors can be specified in this language. By keeping the language simple, we gain some very important properties. The most important is that there is a unique weakest trajectory that satisfies a formula. By focusing initially on the scalar version, we avoid the added complexity of the symbolic case while building a foundation upon which this more general formulation can be based.

Assume $\langle \mathcal{S}, \sqsubseteq \rangle$ is a lattice with universal lower bound $\perp$. Let $\mathcal{P}$ denote a set of simple predicates over $\mathcal{S}$. A *trajectory formula* is defined recursively as:

1. **Simple predicates**: p is a trajectory formula if $p \in \mathcal{P}$.
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 either 0 or 1.
4. **Next time**: $(\mathbf{N}F)$ is a trajectory formula if F is a trajectory formula.

A trajectory formula is said to be *instantaneous* if it contains no next-time operators. Such a formula expresses system properties at only a single point in time. For convenience, we often drop parentheses when the intended precedence is clear. The domain restriction appears at first somewhat strange. Its usefulness will not become apparent until later when we extend the trajectory formulas to a symbolic domain. Observe also that our language does not include either disjunction or negation operations. The motivation and implications of this restriction will be discussed later.

The set of simple predicates is arbitrary. However, for convenience, we will always assume that the predicate $p_0(s) \equiv \mathit{true}$ is in $\mathcal{P}$. Observe that p_0 is indeed a simple predicate with defining value $\perp$.

In switch-level verification the natural simple predicates are of the following form:

1. $(n_i$ **is** $0)$ where $n_i \in \mathcal{N}$, and
2. $(n_i$ **is** $1)$ where $n_i \in \mathcal{N}$.

In other words, our simple predicates ask whether a node in the circuit is known to be 0 or 1. It is easy to see that $(n_i$ **is** $0)$ and $(n_i$ **is** $1)$ are indeed simple with defining values

$$\langle X, \ldots, X, 0, X, \ldots, X \rangle$$

and

$$\langle X, \ldots, X, 1, X, \ldots, X \rangle,$$

where the 0 (1) is in position i. The only somewhat strange property of these predicates is that they are both true in the (artificially introduced) $\top$ state. We ask the reader to simply accept this for the time being. We will discuss the ramifications of this later. For our example circuit of Fig. 2 we will use the five simple predicates: *true*, `in` **is** 0, `in` **is** 1, `out` **is** 0, and `out` **is** 1 with defining values $\langle XX \rangle$, $\langle 0X \rangle$, $\langle 1X \rangle$, $\langle X0 \rangle$, and $\langle X1 \rangle$ respectively.

A trajectory formula describes constraints on some prefix of a trajectory. In order to refer to the length of this prefix, we introduce the concept of "depth" for trajectory formulas. The *depth* of a formula F, written $d(F)$, is defined recursively.

1. $d(p) = 1$ if $p \in \mathcal{P}$ is a simple predicate.
2. $d(F_1 \wedge F_2) = \max(d(F_1), d(F_2))$.
3. $d(e \rightarrow F) = d(F)$.
4. $d(\mathbf{N}F) = 1 + d(F)$.

The depth of a formula is simply the maximum number of nested next time operators plus one.

As a notational convenience, we define for any trajectory formula F

$$F^{[i]} = \begin{cases} F & \text{if } i = 1 \\ F \wedge \mathbf{N}^{d(F)}(F^{[i-1]}) & \text{otherwise,} \end{cases}$$

where $\mathbf{N}^k F$ denotes $(\mathbf{N}(\mathbf{N}(\ldots(F)\ldots)))$ with k next-time operators. This notation allows us to express a condition that repeats over time. For example, the formula $(\texttt{in is } 0)^{[3]}$ states that node `in` stays at 0 for 3 consecutive time units. This is more concise than writing out the formula as $(\texttt{in is } 0) \wedge \mathbf{N}(\texttt{in is } 0) \wedge \mathbf{N}(\mathbf{N}(\texttt{in is } 0))$.

For our example circuit of Fig. 2 we can thus write trajectory formulas like:

$$(\texttt{in is } 0) \wedge \mathbf{N}(\texttt{out is } 1)$$

and

$$(0 \rightarrow ((\texttt{in is } 0) \wedge \mathbf{N}(\texttt{out is } 1))) \wedge (1 \rightarrow ((\texttt{in is } 1) \wedge \mathbf{N}(\texttt{out is } 0))).$$

The truth semantics of a trajectory formula is defined relative to a model structure and a trajectory. In particular, given a model structure $\mathcal{M}$ and a trajectory σ, the truth of a trajectory formula F, written $\sigma \models_{\mathcal{M}} F$, is defined recursively. In the following, assume that both σ and $\sigma^0\tilde{\sigma}$ are members of $L(\mathcal{M})$.

1. $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} p$ iff $p(\sigma^0) = \mathit{true}$.
2. $\sigma \models_{\mathcal{M}} (F_1 \wedge F_2)$ iff $\sigma \models_{\mathcal{M}} F_1$ and $\sigma \models_{\mathcal{M}} F_2$
3. (a) $\sigma \models_{\mathcal{M}} (1 \rightarrow F)$ iff $\sigma \models_{\mathcal{M}} F$
   (b) $\sigma \models_{\mathcal{M}} (0 \rightarrow F)$ holds for every σ.
4. $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} \mathbf{N}F$ iff $\tilde{\sigma} \models_{\mathcal{M}} F$.

For example, given the trajectory $\sigma = \langle 00\rangle\langle 01\rangle\langle XX\rangle\langle XX\rangle \ldots$ for the circuit shown in Fig. 2, it is easy to verify that $\sigma \models_{\mathcal{M}} (\texttt{in is } 0) \wedge \mathbf{N}(\texttt{in is } 0)$, but that

$$\sigma \not\models_{\mathcal{M}} (0 \rightarrow ((\texttt{in is } 0) \wedge \mathbf{N}(\texttt{out is } 1))) \wedge (1 \rightarrow ((\texttt{in is } 1) \wedge \mathbf{N}(\texttt{out is } 0))).$$

5 Properties of Trajectory Formulas

We can extend the definition of simplicity from predicates to formulas in the obvious way, i.e., given a model structure $\mathcal{M}$, a formula F is said to be *simple* iff there is a *defining trajectory* $\overline{\sigma} \in L(\mathcal{M})$ such that $\sigma \models_{\mathcal{M}} F$ iff $\overline{\sigma} \sqsubseteq \sigma$. In this section we first show that trajectory formulas are simple. We then show how the defining sequence can be constructed. The construction is direct and very efficient. As a result, if the main verification task can be phrased in terms of "for every trajectory σ that satisfies the trajectory formula A, verify that the trajectory also satisfies the formula C", it becomes obvious how the verification can be carried out: compute the defining trajectory for the formula A and check that the formula C holds for this trajectory.

Before we can continue, we need a monotonicity result for trajectory formulas. The following lemma states that if a trajectory formula holds for some trajectory σ, then it also holds for every trajectory τ such that $\sigma \sqsubseteq \tau$.

Lemma 1 *If $\sigma, \tau \in L(\mathcal{M})$ and $\sigma \sqsubseteq \tau$ then*

$$\sigma \models_{\mathcal{M}} F \implies \tau \models_{\mathcal{M}} F$$

Proof: We prove the claim by induction on the formula structure. For the basis case, if $F = p$, for some simple predicate $p \in \mathcal{P}$ with defining value $\overline{p}$, then if $\sigma = \sigma^0\tilde{\sigma}$ and $\sigma \models_{\mathcal{M}} F$ it follows from the truth semantics of F that $p(\sigma^0) = \mathit{true}$. By the definition of a simple predicate it thus follows that $\overline{p} \sqsubseteq \sigma^0$. If $\tau = \tau^0\tilde{\tau}$ it follows from the fact that $\sigma \sqsubseteq \tau$ that $\overline{p} \sqsubseteq \sigma^0 \sqsubseteq \tau^0$. Hence, we can conclude that $\tau^0\tilde{\tau} \models_{\mathcal{M}} F$.

If $F = (F_1 \wedge F_2)$ then $\sigma \models_{\mathcal{M}} F$ implies that $\sigma \models_{\mathcal{M}} F_1$ and $\sigma \models_{\mathcal{M}} F_2$. Assuming inductively that the claim holds for the formulas F_1 and F_2, it follows that $\tau \models_{\mathcal{M}} F_1$ and that $\tau \models_{\mathcal{M}} F_2$. This, together with the truth semantics for F, imply that $\tau \models_{\mathcal{M}} F$.

If $F = (1 \rightarrow F_1)$ and $\sigma \models_{\mathcal{M}} F$ then, by the truth semantics, it follows that $\sigma \models_{\mathcal{M}} F_1$. Assuming inductively that the claim holds for F_1, i.e., that $\tau \models_{\mathcal{M}} F_1$, it follows directly that $\tau \models_{\mathcal{M}} F$. On the other hand, if $F = (0 \rightarrow F_1)$ then the claim follows trivially since $(0 \rightarrow F_1)$ holds for every trajectory in $L(\mathcal{M})$.

Finally, if $F = \mathbf{N}F_1$ then, by the truth semantics, $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} F$ implies that $\tilde{\sigma} \models_{\mathcal{M}} F_1$. Assuming inductively that the claim holds for F_1, i.e., that $\tilde{\tau} \models_{\mathcal{M}} F_1$, it follows immediately that $\tau^0\tilde{\tau} \models_{\mathcal{M}} F$. □

Before stating our next result, it is convenient to introduce an infix "choice" function mapping $\{0,1\} \times \mathcal{S}^\omega$ to $\mathcal{S}^\omega$ and which is defined as:

$$e?\delta = \begin{cases} \delta & \text{if } e = 1 \\ \perp\perp \ldots & \text{otherwise} \end{cases}$$

We now show that given a trajectory formula F we can construct its *defining sequence* δ_F. This sequence is the weakest possible in the sense that $\sigma \models_{\mathcal{M}} F$ iff $\delta \sqsubseteq \sigma$. Note that δ_F is not necessarily a trajectory. We define δ_F recursively as follows:

1. $\delta_p = \overline{p}\ \perp\perp \ldots$ if $p \in \mathcal{P}$ is a simple predicate with defining value $\overline{p}$.
2. $\delta_{F_1 \wedge F_2} = \mathit{lub}(\delta_{F_1}, \delta_{F_2})$.
3. $\delta_{e \rightarrow F} = e?\delta_F$.
4. $\delta_{\mathbf{N}F} = \perp\, \delta_F$.

For the particular case of switch-level verification and the model structure $\mathcal{M}^{\mathcal{C}}$, consider the trajectory formula: $F = (\texttt{in is } 0) \wedge \mathbf{N}(\texttt{in is } 0)$. It is straightforward to see that

$$\begin{array}{lcl} \delta_{\texttt{in is } 0} & = & \langle 0X\rangle\ \langle XX\rangle\ \langle XX\rangle\ \ldots \\ \delta_{\mathbf{N}(\texttt{in is } 0)} & = & \langle XX\rangle\ \langle 0X\rangle\ \langle XX\rangle\ \langle XX\rangle\ \ldots \\ \delta_{(\texttt{in is } 0) \wedge \mathbf{N}(\texttt{in is } 0)} & = & \langle 0X\rangle\ \langle 0X\rangle\ \langle XX\rangle\ \langle XX\rangle\ \ldots \end{array}$$

Note that δ_F is not a trajectory as can be seen from Fig. 4. However, it is clearly smaller than several trajectories. For example, $\delta_F \sqsubseteq \langle 0X\rangle\ \langle 01\rangle\ \langle X1\rangle\ \langle XX\rangle\ \ldots$ and $\delta_F \sqsubseteq \langle 0X\rangle\ \langle 01\rangle\ \langle 01\rangle\ \langle X1\rangle\ \langle XX\rangle \ldots$.

Our prohibition of disjunction and negation is partially justified by our desire to have a unique weakest sequence for each formula. For example, we can find a sequence weaker than any other sequence satisfying the formula $F = (\texttt{in is } 0) \vee \mathbf{N}(\texttt{in is } 0)$ by taking the greatest lower bound of the sequences for $(\texttt{in is } 0)$ and $\mathbf{N}(\texttt{in is } 0)$. This would yield the sequence $\langle XX\rangle\ \langle XX\rangle\ \langle XX\rangle\ \ldots$. Unfortunately, this sequence does not itself satisfy F, and hence the verifier would yield overly

pessimistic results. One can also see that there is no unique weakest sequence satisfying the formula $\neg(\texttt{in is } 0)$. More typically, the user really wants to use the formula $(\texttt{in is } 1)$, in any case, and hence this restriction is not as serious as it may initially seem.

In general, we have the following result.

Lemma 2 *For any trajectory formula F let δ_F be constructed as above. Then for every $\sigma \in L(\mathcal{M})$*

$$\sigma \models_{\mathcal{M}} F \iff \delta_F \sqsubseteq \sigma$$

Proof: Assume that $\sigma \in L(\mathcal{M})$, $\sigma \models_{\mathcal{M}} F$, and that $\sigma = \sigma^0\tilde{\sigma}$. We first prove that $\delta_F \sqsubseteq \sigma$ by induction on the formula structure.

For the basis, if $F = p$, for some simple predicate $p \in \mathcal{P}$ with defining value $\overline{p}$, then, by definition, $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} F$ implies that $\overline{p} \sqsubseteq \sigma^0$. Since $\delta_F = \overline{p} \perp\perp \ldots \sqsubseteq \sigma^0\tilde{\sigma} = \sigma$, the basis holds. Thus assume inductively that the claim holds for formulas F_1 and F_2.

If $F = (F_1 \wedge F_2)$ then $\sigma \models_{\mathcal{M}} F$ implies that $\sigma \models_{\mathcal{M}} F_1$ and $\sigma \models_{\mathcal{M}} F_2$. By the induction hypothesis it thus follows that $\delta_{F_1} \sqsubseteq \sigma$ and that $\delta_{F_2} \sqsubseteq \sigma$. Hence, σ is an upper bound on both δ_{F_1} and δ_{F_1}. Consequently, σ is also an upper bound on $lub(\delta_{F_1}, \delta_{F_2})$, i.e., $\delta_F = lub(\delta_{F_1}, \delta_{F_2}) \sqsubseteq \sigma$, and the claim follows.

If $F = (1 \rightarrow F)$ then $\sigma \models_{\mathcal{M}} F$ implies that $\sigma \models_{\mathcal{M}} F_1$, and thus, by the inductive assumption, that $\delta_{F_1} \sqsubseteq \sigma$. However, by definition, $\delta_F = \delta_{F_1}$ and the result follows. On the other hand, if $F = (0 \rightarrow F)$ then $\delta_F = \perp\perp \ldots$ and the result follows trivially.

Finally, if $F = \mathbf{N}F_1$ then $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} F$ implies that $\tilde{\sigma} \models_{\mathcal{M}} F_1$. By Proposition 1 it follows that $\tilde{\sigma} \in L(\mathcal{M})$. Therefore, by the induction hypothesis, it follows that $\delta_{F_1} \sqsubseteq \tilde{\sigma}$. Since $\delta_F = \perp \delta_{F_1} \sqsubseteq \sigma^0\tilde{\sigma}$ the result follows, and the induction step goes through.

Conversely, we now show that if $\sigma = \sigma^0\tilde{\sigma}$ is a trajectory in $L(\mathcal{M})$ and $\delta_F \sqsubseteq \sigma$, then $\sigma \models_{\mathcal{M}} F$. Again, we show this by induction on the structure of F.

For the basis, if $F = p$, for some simple predicate $p \in \mathcal{P}$ with defining value $\overline{p}$, then, by definition, $\delta_F = \overline{p} \perp\perp \ldots$. Since, by assumption, $\delta_F \sqsubseteq \sigma^0\tilde{\sigma}$ it follows that $\overline{p} \sqsubseteq \sigma^0$ and thus that $\sigma \models_{\mathcal{M}} F$ and the basis holds. Hence, assume inductively that $\delta_{F_1} \sqsubseteq \sigma$ and $\delta_{F_2} \sqsubseteq \sigma$ implies $\sigma \models_{\mathcal{M}} F_1$ and $\sigma \models_{\mathcal{M}} F_2$.

If $F = (F_1 \wedge F_2)$ then $\delta_F = lub(\delta_{F_1}, \delta_{F_2})$. This together with the assumption $\delta_F \sqsubseteq \sigma$ and the definition of *lub* imply that $\delta_{F_1} \sqsubseteq \sigma$ and that $\delta_{F_2} \sqsubseteq \sigma$. Hence, by the induction hypothesis, $\sigma \models_{\mathcal{M}} F_1$ and $\sigma \models_{\mathcal{M}} F_2$. By the truth semantics it thus follows that $\sigma \models_{\mathcal{M}} F$.

If $F = (1 \rightarrow F)$ then $\delta_F = \delta_{F_1}$. Since, by assumption, $\delta_F \sqsubseteq \sigma$ it follows that $\delta_{F_1} \sqsubseteq \sigma$. Hence, by the induction hypothesis, it follows that $\sigma \models_{\mathcal{M}} F_1$. Together with the truth semantics we can conclude that $\sigma \models_{\mathcal{M}} F$. On the other hand, if $F = (0 \rightarrow F)$ then then the result holds trivially since $\sigma \models_{\mathcal{M}} F$ holds for every $\sigma \in L(\mathcal{M})$.

Finally, if $F = \mathbf{N}F_1$ then $\delta_F = \perp \delta_{F_1}$. Since, by assumption, $\delta_F \sqsubseteq \sigma = \sigma^0\tilde{\sigma}$ it thus follows that $\delta_{F_1} \sqsubseteq \tilde{\sigma}$ and thus, by the induction hypothesis, that $\tilde{\sigma} \models_{\mathcal{M}} F_1$. Consequently, by the truth semantics, we can conclude that $\sigma \models_{\mathcal{M}} F$ and the induction goes through and the claim follows. □

From the above lemma we know that any trajectory satisfying F must be greater than or equal to its defining sequence δ_F. Thus computing δ_F and then determining if a trajectory is greater than or equal to δ_F allows us to quickly test whether the trajectory satisfies the formula F. However, δ_F is not necessarily itself a trajectory. In the following we will show how to combine the constraints on a state sequence implied by δ_F with those imposed by the system's excitation function to give a trajectory. In fact, we will show that the obtained trajectory is the weakest possible trajectory satisfying F.

It turns out that a slightly more general concept than a defining trajectory is often useful. Thus, assume $\delta_F = \delta_F^0\delta_F^1\ldots$ is the defining sequence for a formula F. Define $\tau_F(z) = \tau_F^0(z)\tau_F^1(z)\ldots$ inductively as follows:

$$\tau_F^i(z) = \begin{cases} lub(\delta_F^0, z) & \text{if } i = 0 \\ lub(\delta_F^i, Y(\tau_F^{i-1}(z))) & \text{otherwise} \end{cases}$$

To illustrate the above construction, let us return to the trajectory formula

$$F = (\texttt{in is } 0) \wedge \mathbf{N}(\texttt{in is } 0)$$

with defining sequence $\delta_F = \langle 0X\rangle\ \langle 0X\rangle\ \langle XX\rangle\ \langle XX\rangle\ \ldots$. Assume we would like to compute $\tau_F(\bot) = \tau_F(XX)$. From the construction above, it follows immediately that

$$\begin{array}{lcl} \tau_F^0(\bot) & = & 0X \\ \tau_F^1(\bot) & = & lub(\delta_F^1, Y(0X)) = lub(0X, X1) = 01 \\ \tau_F^2(\bot) & = & lub(\delta_F^2, Y(01)) = lub(XX, X1) = X1 \\ \tau_F^3(\bot) & = & lub(\delta_F^3, Y(X1)) = lub(XX, XX) = XX \\ \tau_F^i(\bot) & = & XX \quad \text{for } i \geq 4 \end{array}$$

and thus that $\tau_F(\bot) = \langle 0X\rangle\ \langle 01\rangle\ \langle X1\rangle\ \langle XX\rangle\ \langle XX\rangle\ \ldots$. Note that from Fig. 4 we can immediately see that $\tau_F(\bot)$ is a trajectory. It is more difficult to verify, but from Fig. 4 and the truth semantics of F, it can be seen that $\tau_F(\bot)$ is the weakest trajectory that satisfies F and that every other trajectory that satisfies F is greater than $\tau_F(\bot)$. This is in fact no coincidence as we now show.

Before we establish the main properties of $\tau_F(z)$, the following monotonicity property will be needed.

Lemma 3 *If $s \sqsubseteq t$ then $\tau_F(s) \sqsubseteq \tau_F(t)$, for any trajectory formula F.*

Proof: We prove that $\tau_F^i(s) \sqsubseteq \tau_F^i(t)$ by induction on i. For the base case we have that

$$\tau_F^0(s) = lub(s, \delta_F^0) \sqsubseteq lub(t, \delta_F^0) = \tau_F^0(s)$$

by the monotonicity of *lub*. Assume now inductively that $\tau_F^i(s) \sqsubseteq \tau_F^i(t)$ for some $i \geq 0$. It follows from the definition of $\tau_F^{i+1}(z)$, the induction hypothesis, and the monotonicity of *lub* and Y that $\tau_F^{i+1}(s) = lub(\delta_F^{i+1}, Y(\tau_F^i(s))) \sqsubseteq lub(\delta_F^{i+1}, Y(\tau_F^i(t))) = \tau_F^{i+1}(t)$ and the claim follows. □

The second key lemma of this section states that there is a defining trajectory for every trajectory formula F and start condition z. More formally:

Lemma 4 *Assume $\tau_F(z)$ is defined as above, then:*

1. $\tau_F(z) \in L(\mathcal{M}, z)$,
2. $\tau_F(z) \models_{\mathcal{M}} F$, *and*
3. *for every* $\sigma \in L(\mathcal{M}, z)$

$$\sigma \models_{\mathcal{M}} F \iff \tau_F(z) \sqsubseteq \sigma$$

Proof: In order to prove that $\tau_F(z) \in L(\mathcal{M}, z)$ it is sufficient to show that $z \sqsubseteq \tau_F^0(z)$ and that $Y(\tau_F^{i-1}(z)) \sqsubseteq \tau_F^i(z)$ for $i \geq 1$. Since $\tau_F^0(z) = lub(z, \delta_F^0)$, we can immediately conclude that $z \sqsubseteq \tau_F^0(z)$. On the other hand, by the definition of *lub* it follows that for $i \geq 1$,

$$Y(\tau_F^{i-1}(z)) \sqsubseteq lub(\delta_F^i, Y(\tau_F^{i-1}(z))).$$

However $\tau_F^i(z) = lub(\delta_F^i, Y(\tau_F^{i-1}(z)))$, and thus $Y(\tau_F^{i-1}(z)) \sqsubseteq \tau_F^i(z)$ for $i \geq 1$. Altogether, $\tau_F(z) \in L(\mathcal{M}, z)$.

By the definition of *lub* it also follows that

$$\delta_F^i \sqsubseteq lub(\delta_F^i, Y(\tau_F^{i-1}(z))) = \tau_F^i(z) \quad \text{for } i \geq 1.$$

Hence, $\delta_F \sqsubseteq \tau_F(z)$. This, together with the fact that $\tau_F(z) \in L(\mathcal{M}, z) \subseteq L(\mathcal{M})$, means that Lemma 2 apply. Thus, $\tau_F(z) \models_{\mathcal{M}} F$.

Now assume $\sigma \in L(\mathcal{M}, z)$. Since $\tau_F(z)$ is a trajectory and $\tau_F(z) \models_{\mathcal{M}} F$ we can apply Lemma 1. Hence, if $\tau_F(z) \sqsubseteq \sigma$ then $\sigma \models_{\mathcal{M}} F$.

Finally, we establish the converse by showing that for any $\sigma \in L(\mathcal{M}, z)$, $\tau_F(z) \sqsubseteq \sigma$. Thus, assume $\sigma = \sigma^0\sigma^1 \ldots$ is a trajectory, $z \sqsubseteq \sigma^0$, and that $\sigma \models_{\mathcal{M}} F$. We prove by induction on i that $\tau_F^i(z) \sqsubseteq \sigma^i$.

Since $\sigma = \sigma^0\tilde{\sigma}$ is a trajectory, Lemma 2 applies. Consequently, $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} F$ implies that $\delta_F = \delta_F^0\delta_F^1 \ldots \sqsubseteq \sigma^0\tilde{\sigma} = \sigma$. Furthermore, since $\sigma = \sigma^0\tilde{\sigma} \in L(\mathcal{M}, z)$ it follows that $z \sqsubseteq \sigma^0$. In other words, σ^0 is an upper bound for both z and δ_F^0 and thus $lub(z, \delta_F^0) \sqsubseteq \sigma^0$. However, since $\tau_F^0(z) = lub(z, \delta_F^0)$ it follows directly that $\tau_F^0(z) \sqsubseteq \sigma^0$ and the basis case holds.

Now assume inductively that $\tau_F^i(z) \sqsubseteq \sigma^i$ for some $i > 0$. Since σ is a trajectory, it follows that $Y(\sigma^i) \sqsubseteq \sigma^{i+1}$. Also, by Lemma 2 we know that $\delta_F \sqsubseteq \sigma$ and thus that $\delta_F^{i+1} \sqsubseteq \sigma^{i+1}$. Together, these facts imply that σ^{i+1} is an upper bound to both $Y(\sigma^i)$ and δ_F^{i+1}. Consequently, $lub(\delta_F^{i+1}, Y(\sigma^i)) \sqsubseteq \sigma^{i+1}$. However, by the induction hypothesis, $\tau_F^i(z) \sqsubseteq \sigma^i$. Hence, by the monotonicity of Y and *lub*, it follows that

$$\tau_F^{i+1}(z) = lub(\delta_F^{i+1}, Y(\tau_F^i(z))) \sqsubseteq lub(\delta_F^{i+1}, Y(\sigma^i)) \sqsubseteq \sigma^{i+1}$$

and the induction step goes through and the lemma follows. □

Another way of stating this lemma is that every trajectory formula F is simple with defining trajectory $\tau_F(\bot)$.

The above lemmas give a simple method for computing the defining trajectory and the defining sequence for a trajectory formula. Unfortunately, there is a practical difficulty, since both the defining trajectory and the defining sequence are theoretically infinite sequences. The following technical lemma will be useful later to show that only a finite prefix of the defining trajectories and sequences are needed.

Lemma 5 *Let F be a trajectory formula and let $\delta_F = \delta_F^0\delta_F^1 \ldots$ be the defining sequence for formula F. Then $\delta_F^i = \bot$ for $i \geq d(F)$.*

Proof: We prove the claim by induction on the formula structure. For the basis, if $F = p$, for some simple predicate p with defining value $\overline{p}$, then $\delta_F = \overline{p} \bot\bot \ldots$. Since, $d(p) = 1$, it follows directly that $\delta_F^i = \bot$ for $i \geq d(F)$ and the basis holds.

Assume inductively that $\delta_{F_1}^i = \bot$ for $i \geq d(F_1)$ and that $\delta_{F_2}^i = \bot$ for $i \geq d(F_2)$ for some trajectory formulas F_1 and F_2. If $F = F_1 \wedge F_2$ then $d(F) = \max(d(F_1), d(F_2))$. Consider any $i \geq d(F)$. Since $d(F) \geq d(F_1)$ and $d(F) \geq d(F_2)$ it follows from the induction hypothesis that $\delta_{F_1}^i = \bot$ and that $\delta_{F_2}^i = \bot$. Furthermore, since $\delta_F = lub(\delta_{F_1}, \delta_{F_2})$ we can conclude that $\delta_F^i = \bot$.

If $F = e \rightarrow F_1$ then there are two cases to consider. If $e = 0$ then $\delta_F = \bot\bot \ldots$ and the claim follows trivially. On the other hand, if $e = 1$ then $\delta_F = \delta_{F_1}$. By the induction hypothesis, $\delta_{F_1}^i = \bot$ for every $i \geq d(F_1)$. Since, $d(F) = d(F_1)$, we can conclude that $\delta_F^i = \bot$ for every $i \geq d(F)$.

Finally, if $F = \mathbf{N}F_1$ then $\delta_F = \bot\ \delta_{F_1}$. By the induction hypothesis, $\delta_{F_1}^i = \bot$ for every $i \geq d(F_1)$. Consequently, $\delta_F^i = \bot$ for every $i \geq d(F_1) + 1$. However, $d(F) = 1 + d(F_1)$ and thus $\delta_F^i = \bot$ for every

$i \geq d(F)$. □

From this result we immediately get the following corollary.

Corollary 1 *Assume A and C are two trajectory formulas. Let $\tau_A = \tau_A^0 \tau_A^1 \ldots$ be the defining trajectory for formula A and let $\delta_C = \delta_C^0 \delta_C^1 \ldots$ be the defining sequence for formula C. Then*

$$\delta_C \sqsubseteq \tau_A \quad \textit{iff} \quad \delta_C^i \sqsubseteq \tau_A^i \textit{ for } 0 \leq i < d(C)$$

6 Verification Methodology

Our specification language describes a property of the system $\mathcal{M}$ as a "trajectory assertion". Again, we have chosen a quite limited language in order to gain efficiency. We have three types of constructs: simple assertions, sequences, and iterations. Simple assertions are of the form "if the system ever goes through a sequence of states satisfying trajectory formula A, then the sequence of states better also satisfy the trajectory formula C". Sequences of assertions allow representing system behaviors that shift from one "mode" to another. For example, it is convenient to use in describing the desired behavior during each clock cycle for a microprocessor during the execution of a multi-cycle instruction. Finally, a simple assertion can also be iterated an arbitrary number of times. This construct is primarily useful for, automatically, establishing and proving invariants of the system. For example, a typical use of the iteration construct is when specifying the possibility of an arbitrary number of wait-states in a microprocessor. More specifically, we may want to verify that the processor works correctly no matter how many wait-states the external memory interface imposes. This could be accomplished by describing the constraints on the inputs during "wait cycles" and iterate this simple assertion an arbitrary number of times.

More formally, a *trajectory assertion* is defined recursively as:

1. **Simple assertions:** $[A \Longrightarrow C]$, where A and C are trajectory formulas and $d(A) = d(C)$.
2. **Sequences:** $[A \Longrightarrow C]\,;G_1$, where A and C are trajectory formulas, $d(A) = d(C)$, and G_1 is a trajectory assertion.
3. **Iterations:** $[A \Longrightarrow C]^*\,;G_1$, where A and C are trajectory formulas, $d(A) = d(C)$, and G_1 is a trajectory assertion.

A trajectory assertion that does not contain any iteration, is said to be *iteration-free*.

The definition of a trajectory assertion is somewhat restrictive. For example, it does not allow a trajectory assertion to end with an iteration. The reason for this restriction is to simplify the definition of the truth semantics of trajectory assertions. In practice, it turns out not to be a serious restriction since one can always append $[\mathit{true} \Longrightarrow \mathit{true}]$ to an assertion that otherwise would end with an iteration.

To illustrate trajectory assertions, consider first our inverter circuit of Fig. 2. The following two assertions can constitute our specification of a unit-delay inverter:

$$[\texttt{in is } 0 \wedge \mathbf{N}\mathit{true} \Longrightarrow \mathbf{N}\texttt{out is } 1]$$

and

$$[\texttt{in is } 1 \wedge \mathbf{N}\mathit{true} \Longrightarrow \mathbf{N}\texttt{out is } 0]\,.$$

Note that the $\mathbf{N}\mathit{true}$ parts in the antecedents are simply there in order to make the depth of the antecedent equal the depth of the consequent. In a practical system, these "filler" functions

would be added automatically by the verification system and thus would not have to be expressed explicitly. However, in order to simplify the presentation of the general theory we have opted to require the depth of the antecedent to be equal to the depth of the consequent.

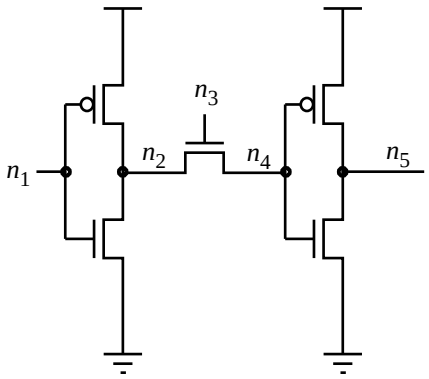

Figure 5: Switch-level latch.

Our next example shows the use of the sequence construct. Consider the switch-level circuit shown in Fig. 5. Intuitively, n_1 is the input to a latch, n_3 is the clock signal, n_4 is the electrical node that stores the state when the clock is low, and n_5 is the output of the output buffer. If the state of the circuit currently is $t \in \mathcal{T}^5$, a typical switch-level analysis of the circuit would derive the excitation functions:

$$y_1(t) = X \quad y_2(t) = \overline{t_1} \quad y_3(t) = X \quad y_4(t) = \overline{t_1}t_4 + t_3\overline{t_1} + \overline{t_3}t_4 \quad y_5(t) = \overline{t_4}$$

where all operators are assumed to be ternary. That is, nodes n_1 and n_3, being input nodes, have excitation X. Nodes n_2 and n_5 are the outputs of simple inverters. Depending on the control signal on n_3, node n_4 will either retain its stored charge ($t_3 = 0$), or get the value from the first inverter ($t_3 = 1$). If $t_3 = X$, node n_4 will have a binary excitation only if the inverter output matches the value already on the node, and value X otherwise. Such excitation functions can be derived automatically from the transistor representation of the circuit by symbolic circuit analysis [8].

Since the latch is a sequential circuit and the clock signal changes the behavior quite drastically, it is natural to specify the desired behavior as a sequence of sub-behaviors—one for each clock phase. For example, the following assertion expresses the desired behavior of the circuit when each clock phase has a duration of two time units:

$$G_1 = \left[((n_1 \textbf{ is } 1) \wedge (n_3 \textbf{ is } 1))^{[2]} \Longrightarrow true^{[2]}\right]; \left[(n_3 \textbf{ is } 0)^{[2]} \Longrightarrow (n_5 \textbf{ is } 1)^{[2]}\right].$$

Recall that $F^{[2]} = F \wedge \mathbf{N}F$ for an instantaneous trajectory formula F.

The above assertion only verifies the circuit behavior for one particular clock timing. In general, the desired behavior of a latch can be expressed informally as: "given that the clock cycle is longer than some minimum time, the circuit can load an input when the clock is high and retain it when the clock goes low". The iteration construct can be used to formulate such a specification, yielding the assertion:

$$\begin{aligned} G_2 \quad = \quad & \left[((n_1 \textbf{ is } 1) \wedge (n_3 \textbf{ is } 1))^{[2]} \Longrightarrow true^{[2]}\right]; [(n_1 \textbf{ is } 1) \wedge (n_3 \textbf{ is } 1) \Longrightarrow true]^*; \\ & [(n_3 \textbf{ is } 0) \Longrightarrow (n_5 \textbf{ is } 1)]^*; [true \Longrightarrow true]. \end{aligned}$$

Intuitively, we are here stating that if the clock is high and the input is 1 for at least two time units and then the clock goes low, the output will remain 1. Note that any circuit passing G_2 will pass G_1, but the opposite does not necessarily hold.

The example above illustrates our motivation for introducing the iteration construct. It allows the verification of systems that are characterized by periods of activity interspersed by periods in which the circuit is waiting for some external event before continuing. With a single trajectory evaluation we can verify correct circuit behavior for all possible durations of these idle periods. In the above example, the external events correspond to transitions of the clock, with the circuit remaining in a stable state until the clock changes. The antecedent of the iteration construct is an instantaneous formula specifying inputs that will be held fixed for the remainder of the clock phase, while the consequent specifies state values that should remain stable as long as the clock and inputs are held fixed. In other cases the iteration construct includes temporal operators indicating some periodic behavior of the system, such the cycling of clocks while a processor is in a wait state. Note that this construct should not be confused with the operators of temporal logic denoting "eventual" behavior, such as the **F** or **U** operators of CTL [16]. We require the consequent of the iteration construct to hold for any number of repetitions (including 0) as long as the antecendent is satisfied.

Before we define the truth semantics of a trajectory assertion we need to introduce a function that removes some of the first elements in a sequence. Let the *suffix* of a sequence σ be defined recursively as follows:

$$\mathit{suffix}(n, \sigma^0\tilde{\sigma}) = \begin{cases} \sigma^0\tilde{\sigma} & \text{if } n = 0 \\ \mathit{suffix}(n-1, \tilde{\sigma}) & \text{otherwise.} \end{cases}$$

Intuitively, the suffix function applied to some sequence removes the first n elements in the sequence.

The truth semantics of a trajectory assertion is defined relative to a model structure and a set of trajectories in this model structure. In particular, given a model structure $\mathcal{M}$ and a set L of trajectories, the truth of a trajectory assertion G, written $L \models_{\mathcal{M}} G$, is defined recursively as follows:

1. $L \models_{\mathcal{M}} [A \Longrightarrow C]$ holds iff $\sigma \models_{\mathcal{M}} A$ implies $\sigma \models_{\mathcal{M}} C$ for all $\sigma \in L$.
2. $L \models_{\mathcal{M}} [A \Longrightarrow C]\,;G_1$ holds iff $L \models_{\mathcal{M}} [A \Longrightarrow C]$ and $\tilde{L} \models_{\mathcal{M}} G_1$, where
$$\tilde{L} = \{\tilde{\sigma} \mid \tilde{\sigma} = \mathit{suffix}(d(A), \sigma),\ \sigma \in L \text{ and } \sigma \models_{\mathcal{M}} A\}.$$
3. $L \models_{\mathcal{M}} [A \Longrightarrow C]^*\,;G_1$ holds iff $L \models_{\mathcal{M}} G_1$ and $\forall i \geq 1.\ L \models_{\mathcal{M}} \left[A^{[i]} \Longrightarrow C^{[i]}\right];G_1$.

Since we often require a trajectory assertion to hold for all possible trajectories, we use the shorthand $\models_{\mathcal{M}} G$ to denote $L(\mathcal{M}) \models_{\mathcal{M}} G$.

Returning to our examples of trajectory assertions above, we can easily see from Fig. 4 that

$$L(\mathcal{M}^{\mathcal{C}}) \models_{\mathcal{M}} [\texttt{in is } 0 \wedge \mathbf{N}\mathit{true} \Longrightarrow \mathbf{Nout\ is}\ 1],$$

and that

$$L(\mathcal{M}^{\mathcal{C}}) \models_{\mathcal{M}} [\texttt{in is } 1 \wedge \mathbf{N}\mathit{true} \Longrightarrow \mathbf{Nout\ is}\ 0].$$

What we will show in this section is how to determine the validity of a trajectory assertion without having to compute the complete state space as was done in Fig. 4.

The following, rather technical, lemma will be useful later.

Lemma 6 *Given a model structure $\mathcal{M}$, an initial state $z \in \mathcal{S}$, and a trajectory formula F with defining trajectory $\tau_F^0(z)\tau_F^1(z)\ldots$, let $\tilde{L} = \{\tilde{\sigma} \mid \tilde{\sigma} = \mathit{suffix}(d(F), \sigma),\ \sigma \in L(\mathcal{M}, z)$ and $\sigma \models_{\mathcal{M}} F\}$. Then $\tilde{L} = L(\mathcal{M}, \tau_F^{d(F)}(z))$.*

Proof: Assume first that $\tilde{\sigma} \in \tilde{L}$. This implies that there is a $\sigma \in L(\mathcal{M}, z)$ such that $\sigma \models_{\mathcal{M}} F$ and $\tilde{\sigma} = \mathit{suffix}(d(F), \sigma)$. Since $\tilde{\sigma} = \mathit{suffix}(d(F), \sigma)$ and $\sigma \in L(\mathcal{M}, z) \subseteq L(\mathcal{M})$ we can conclude from Proposition 1 that $\tilde{\sigma} \in L(\mathcal{M})$. Hence, in order to prove that $\tilde{\sigma} \in L(\mathcal{M}, \tau_F^{d(F)}(z))$ it suffices to show that $\tau_F^{d(F)}(z) \sqsubseteq \tilde{\sigma}^0$. By Lemma 4 we know that $\tau_F(z) \sqsubseteq \sigma$ iff $\sigma \models_{\mathcal{M}} F$ for all $\sigma \in L(\mathcal{M}, z)$. In particular, $\tau_F^{d(F)}(z) \sqsubseteq \sigma^{d(F)} = \tilde{\sigma}^0$ and the claim follows.

Conversely, assume $\tilde{\sigma} \in L(\mathcal{M}, \tau_F^{d(F)}(z))$. Define $\sigma = \sigma^0\sigma^1 \ldots$ as follows:

$$\sigma^i = \begin{cases} \tau_F^i(z) & \text{if } i < d(F) \\ \tilde{\sigma}^{i-d(F)} & \text{otherwise} \end{cases}$$

Clearly $\mathit{suffix}(d(F), \sigma) = \tilde{\sigma}$. If we now can show that $\sigma \in L(\mathcal{M}, z)$ and that $\sigma \models_{\mathcal{M}} F$ it would follow that $\tilde{\sigma} \in \tilde{L}$ and the claim of the lemma would be established.

In order to prove that $\sigma \in L(\mathcal{M}, z)$ we must establish that $z \sqsubseteq \sigma^0$ and that $Y(\sigma^i) \sqsubseteq \sigma^{i+1}$ for $i \geq 0$. To show the former, note that, by definition, $d(F) > 0$ and thus $\sigma^0 = \tau_F^0(z) = \mathit{lub}(z, \delta_F^0)$ and therefore $z \sqsubseteq \sigma^0$. In order to prove the latter we need to consider three cases. If $0 \leq i \leq d(F) - 2$, then $\sigma^{i+1} = \tau_F^{i+1}(z) = \mathit{lub}(\delta_F^{i+1}, Y(\sigma^i))$ and thus $Y(\sigma^i) \sqsubseteq \sigma^{i+1}$. On the other hand, if $i \geq d(F)$ then $\sigma^{i+1} = \tilde{\sigma}^{i+1-d(F)}$ and $\sigma^i = \tilde{\sigma}^{i-d(F)}$. Since $\tilde{\sigma} \in L(\mathcal{M}, \tau_F^{d(F)}(z)) \subseteq L(\mathcal{M})$ it follows that $Y(\sigma^i) = Y(\tilde{\sigma}^{i-d(F)}) \sqsubseteq \tilde{\sigma}^{i+1-d(F)} = \sigma^{i+1}$. Finally, since $\tilde{\sigma} \in L(\mathcal{M}, \tau_F^{d(F)}(z))$ it follows that $\tau_F^{d(F)}(z) \sqsubseteq \tilde{\sigma}^0 = \sigma^{d(F)}$. This, together with the fact that $\tau_F^{d(F)}(z) = \mathit{lub}(\delta_F^{d(F)}, Y(\tau_F^{d(F)-1}(z)))$, implies that $Y(\sigma^{d(F)-1}) = Y(\tau_F^{d(F)-1}(z)) \sqsubseteq \tau_F^{d(F)}(z) \sqsubseteq \sigma^{d(F)}$. Altogether, $Y(\sigma^i) \sqsubseteq \sigma^{i+1}$ for $i \geq 0$ and thus $\sigma \in L(\mathcal{M}, z)$.

By Lemma 4 we know that $\tau_F(z) \models_{\mathcal{M}} F$. If we can prove that $\tau_F(z) \sqsubseteq \sigma$ then, by Lemma 1, it would follow that $\sigma \models_{\mathcal{M}} F$. We prove that $\tau_F^i(z) \sqsubseteq \sigma^i$ for $i \geq 0$ by induction on i. For the basis, $\sigma^0 = \tau_F^0(z)$ and the claim holds trivially. Now assume inductively that the claim holds for some $i - 1 \geq 0$ and consider i. There are three cases to consider. If $0 \leq i \leq d(F) - 1$ then $\sigma^i = \tau_F^i(z)$ and the claim follows trivially. On the other hand, if $i = d(F)$ then $\sigma^{d(F)} = \tilde{\sigma}^0$. Since $\tilde{\sigma} \in L(\mathcal{M}, \tau_F^{d(F)}(z))$ it follows that $\tau_F^{d(F)}(z) \sqsubseteq \tilde{\sigma}^0$ and the claim follows. Finally, if $i > d(F)$ then $\tau_F^i(z) = \mathit{lub}(\delta_F^i, Y(\tau_F^{i-1}))$. However, by Lemma 5, $\delta_F^i = \bot$ for $i > d(F)$. Consequently, $\tau_F^i(z) = Y(\tau_F^{i-1})$. Since we already has established that $\sigma \in L(\mathcal{M}, z) \subseteq L(\mathcal{M})$ it follows that $Y(\sigma^{i-1}) \sqsubseteq \sigma^i$. This, together with the induction hypothesis and the monotonicity of Y, implies that $\tau_F^i(z) = Y(\tau_F^{i-1}) \sqsubseteq Y(\sigma^{i-1}) \sqsubseteq \sigma^i$. In all cases the induction step goes through and the claim follows. □

From the above lemma and the definition of $L(\mathcal{M}, z)$ the following proposition follows directly.

Proposition 2 *Given a model structure $\mathcal{M}$, an initial state z, and a trajectory assertion G, the validity of $L(\mathcal{M}, z) \models_{\mathcal{M}} G$ can be computed recursively as follows:*

1. $L(\mathcal{M}, z) \models_{\mathcal{M}} [A \Longrightarrow C]$ *holds iff* $\sigma \models_{\mathcal{M}} A$ *implies* $\sigma \models_{\mathcal{M}} C$ *for all* $\sigma \in L(\mathcal{M}, z)$.
2. $L(\mathcal{M}, z) \models_{\mathcal{M}} [A \Longrightarrow C]; G_1$ *holds iff* $L(\mathcal{M}, z) \models_{\mathcal{M}} [A \Longrightarrow C]$ *and* $L(\mathcal{M}, \tau_A^{d(A)}(z)) \models_{\mathcal{M}} G_1$.
3. $L(\mathcal{M}, z) \models_{\mathcal{M}} [A \Longrightarrow C]^*; G_1$ *holds iff* $L(\mathcal{M}, z) \models_{\mathcal{M}} G_1$ *and for all* $i \geq 1$*:*

$$L(\mathcal{M}, z) \models_{\mathcal{M}} \left[A^{[i]} \Longrightarrow C^{[i]}\right]; G_1.$$

In view of the properties of defining sequences and trajectories derived in the previous section, our main verification method is captured in the following "satisfaction" predicate for trajectory assertions. The predicate is defined recursively as:

1. $SAT(z, [A \Longrightarrow C])$ iff $\delta_C \sqsubseteq \tau_A(z)$.
2. $SAT(z, [A \Longrightarrow C]; G_1)$ iff $SAT(z, [A \Longrightarrow C])$ and $SAT(\tau_A^{d(A)}(z), G_1)$.
3. $SAT(z, [A \Longrightarrow C]^*; G_1)$ iff $SAT(\tilde{z}, G_1)$ and $SAT(\tilde{z}, [A \Longrightarrow C])$, where
$$\tilde{z} = Gfp\ \xi.\ glb(z, \tau_A^{d(A)}(\xi)).$$

The greatest fixed-point above is well defined and can be computed iteratively since the domain $\mathcal{S}$ is a finite lattice and $glb(z, \tau_A^{d(A)}(\xi))$ is monotone in ξ.

Again returning to our inverter example, we will illustrate the computation of
$$SAT(\bot, [(\texttt{in is } 0) \wedge \mathbf{N}\text{true} \Longrightarrow \mathbf{N}(\texttt{out is } 1)]).$$
First, from Section 5 we get that
$$\delta_{\mathbf{N}(\texttt{out is } 1)} = \langle XX\rangle\ \langle X1\rangle\ \langle XX\rangle\ \langle XX\rangle\ \ldots$$
and that
$$\tau_{(\texttt{in is } 0)\wedge\mathbf{N}true}(\bot) = \langle 0X\rangle\ \langle X1\rangle\ \langle XX\rangle\ \ldots.$$
Consequently, we have $\delta_{\mathbf{N}(\texttt{out is } 1)} \sqsubseteq \tau_{(\texttt{in is } 0)\wedge\mathbf{N}true}(\bot)$ and, from the definition of $\sqsubseteq$ and SAT, that $SAT(\bot, [(\texttt{in is } 0) \wedge \mathbf{N}\text{true} \Longrightarrow \mathbf{N}(\texttt{out is } 1)])$ holds.

To illustrate the computation of SAT for a more complex trajectory assertion, consider again the circuit shown in Fig. 5 and the assertion
$$G_1 = \left[((n_1 \texttt{ is } 1) \wedge (n_3 \texttt{ is } 1))^{[2]} \Longrightarrow true^{[2]}\right]; \left[(n_3 \texttt{ is } 0)^{[2]} \Longrightarrow (n_5 \texttt{ is } 1)^{[2]}\right].$$
For convenience, let $A_1 = ((n_1 \texttt{ is } 1) \wedge (n_3 \texttt{ is } 1))^{[2]}$, $C_1 = true^{[2]}$, $A_2 = (n_3 \texttt{ is } 0)^{[2]}$, and $C_2 = (n_5 \texttt{ is } 1)^{[2]}$. Note that $d(A_1) = d(C_1) = 2$ and $d(A_2) = d(C_2) = 2$. In order to compute $SAT(\bot, G_1)$, we first compute $\tau_{A_1}(\bot) = \tau_{A_1}(\langle XXXXX\rangle)$. From the definition of defining sequence, we get that
$$\delta_{A_1} = \langle 1X1XX\rangle\langle 1X1XX\rangle\langle XXXXX\rangle\langle XXXXX\rangle\ldots$$
and thus
$$\begin{aligned}
&\tau_{A_1}^0(\langle XXXXX\rangle) = lub(\delta_{A_1}^0, \langle XXXXX\rangle) = \langle 1X1XX\rangle\\
&\tau_{A_1}^1(\langle XXXXX\rangle) = lub(\delta_{A_1}^1, Y(\langle 1X1XX\rangle)) = lub(\langle 1X1XX\rangle, \langle X0X0X\rangle) = \langle 1010X\rangle\\
&\tau_{A_1}^2(\langle XXXXX\rangle) = lub(\delta_{A_1}^2, Y(\langle 1010X\rangle)) = lub(\langle XXXXX\rangle, \langle X0X01\rangle) = \langle X0X01\rangle\\
&\tau_{A_1}^3(\langle XXXXX\rangle) = lub(\delta_{A_1}^3, Y(\langle X0X01\rangle)) = lub(\langle XXXXX\rangle, \langle XXXX1\rangle) = \langle XXXX1\rangle\\
&\tau_{A_1}^4(\langle XXXXX\rangle) = lub(\delta_{A_1}^4, Y(\langle XXXX1\rangle)) = lub(\langle XXXXX\rangle, \langle XXXXX\rangle) = \langle XXXXX\rangle\\
&\tau_{A_1}^i(\langle XXXXX\rangle) = \langle XXXXX\rangle \quad \text{for } i \geq 5.
\end{aligned}$$
In particular, $\tau_{A_1}^{d(A_1)}(\bot) = \langle X0X01\rangle$. Also, since $C_1 = true^{[2]}$, and thus $\delta_{C_1} = \bot\bot\ldots$, it follows that $\delta_{C_1} \sqsubseteq \tau_{A_1}(\bot)$ and therefore that $SAT(\bot, [A_1 \Longrightarrow C_1])$ holds. Similarly, we get
$$\delta_{A_2} = \langle XX0XX\rangle\langle XX0XX\rangle\langle XXXXX\rangle\langle XXXXX\rangle\ldots$$
Since, $\tau_{A_1}^{d(A_1)}(\bot) = \langle X0X01\rangle$, we get that $\tau_{A_2}(\tau_{A_1}^{d(A_1)}(\bot)) = \tau_{A_2}(\langle X0X01\rangle)$ equals
$$\begin{aligned}
&\tau_{A_2}^0(\langle X0X01\rangle) = lub(\delta_{A_2}^0, \langle X0X01\rangle) = \langle X0001\rangle\\
&\tau_{A_2}^1(\langle X0X01\rangle) = lub(\delta_{A_2}^1, Y(\langle X0001\rangle)) = lub(\langle XX0XX\rangle, \langle XXX01\rangle) = \langle XX001\rangle\\
&\tau_{A_2}^2(\langle X0X01\rangle) = lub(\delta_{A_2}^2, Y(\langle XX001\rangle)) = lub(\langle XXXXX\rangle, \langle XXX01\rangle) = \langle XXX01\rangle\\
&\tau_{A_2}^3(\langle X0X01\rangle) = lub(\delta_{A_2}^3, Y(\langle XXX01\rangle)) = lub(\langle XXXXX\rangle, \langle XXXX1\rangle) = \langle XXXX1\rangle\\
&\tau_{A_2}^4(\langle X0X01\rangle) = lub(\delta_{A_2}^4, Y(\langle XXXX1\rangle)) = lub(\langle XXXXX\rangle, \langle XXXXX\rangle) = \langle XXXXX\rangle\\
&\tau_{A_2}^i(\langle X0X01\rangle) = \langle XXXXX\rangle \quad \text{for } i \geq 5.
\end{aligned}$$

Since

$$\delta^0_{C_2} = \langle XXXX1\rangle\langle XXXX1\rangle\langle XXXXX\rangle\langle XXXXX\rangle\ldots$$

it follows immediately that $SAT(\langle \mathrm{X0X01}\rangle, [A_2 \Longrightarrow C_2])$ holds. Altogether, we have that

$$SAT(\bot, [A_1 \Longrightarrow C_1]\,;[A_2 \Longrightarrow C_2]).$$

Finally, we illustrate the computation of SAT for an assertion containing an iteration by computing $SAT(\bot, G_2)$, where

$$\begin{aligned} G_2 \;=\; & \Big[((n_1 \textbf{ is } 1) \wedge (n_3 \textbf{ is } 1))^{[2]} \Longrightarrow true^{[2]}\Big]\,;[(n_1 \textbf{ is } 1) \wedge (n_3 \textbf{ is } 1) \Longrightarrow true]^*\,; \\ & [(n_3 \textbf{ is } 0) \Longrightarrow (n_5 \textbf{ is } 1)]^*\,;[true \Longrightarrow true]\,. \end{aligned}$$

Again for convenience, let $A_1 = ((n_1 \textbf{ is } 1) \wedge (n_3 \textbf{ is } 1))^{[2]}$, $C_1 = true^{[2]}$, $A_2 = ((n_1 \textbf{ is } 1) \wedge (n_3 \textbf{ is } 1))$, $C_2 = true$, $A_3 = (n_3 \textbf{ is } 0)$, and $C_3 = (n_5 \textbf{ is } 1)$. As above, we get that $SAT(\bot, [A_1 \Longrightarrow C_1])$ holds and that $\tau^{d(A_1)}_{A_1}(\bot) = \langle X0X01\rangle$. We now must compute the greatest fixed point value to represent the set of all reachable states after some iterations matching A_2, i.e., we need to compute

$$Gfp\;\xi.\; glb(\langle X0X01\rangle, \tau^{d(A_2)}_{A_2}(\xi)).$$

We do this by iterating starting from $\top$. Note that $Y(\top) = \top$ and thus $\tau^i_A(\top) = \top$ for all trajectory formulas A and $i \geq 1$. Thus:

$$\begin{aligned} \xi^0 &= \top \\ \xi^1 &= glb(\langle X0X01\rangle, \tau^{d(A_2)}_{A_2}(\xi^0)) = glb(\langle X0X01\rangle, \tau^{d(A_2)}_{A_2}(\top)) = \langle X0X01\rangle \\ \xi^2 &= glb(\langle X0X01\rangle, \tau^{d(A_2)}_{A_2}(\xi^1)) \\ &= glb(\langle X0X01\rangle, \tau^{d(A_2)}_{A_2}(\langle X0X01\rangle)) \\ &= glb(\langle X0X01\rangle, \langle X0X01\rangle) = \langle X0X01\rangle \end{aligned}$$

and thus $\tilde{z} = Gfp\;\xi.\; glb(\langle X0X01\rangle, \tau^{d(A_2)}_{A_2}(\xi)) = \langle X0X01\rangle$. Since $C_2 = true$, and thus $\delta_{C_2} = \bot\bot\ldots$ it follows immediately that $SAT(\tilde{z}, [A_2 \Longrightarrow C_2])$. These computations indicate that the circuit was already in the stable state $\langle X0X01\rangle$ after the first 2 unit steps and will remain in this state as long as n_1 and n_3 are held at 1.

In a similar fashion, we now compute the fixed point for the set of reachable states after some iterations of A_3. In other words, we compute

$$Gfp\;\xi.\; glb(\langle X0X01\rangle, \tau^{d(A_3)}_{A_3}(\xi)).$$

Here we get:

$$\begin{aligned} \chi^0 &= \top \\ \chi^1 &= glb(\langle X0X01\rangle, \tau^{d(A_3)}_{A_3}(\chi^0)) = glb(\langle X0X01\rangle, \tau^{d(A_3)}_{A_3}(\top)) \\ &= glb(\langle X0X01\rangle, \top) = \langle X0X01\rangle \\ \chi^2 &= glb(\langle X0X01\rangle, \tau^{d(A_3)}_{A_3}(\chi^1)) = glb(\langle X0X01\rangle, \tau^{d(A_3)}_{A_3}(\langle X0X01\rangle)) \\ &= glb(\langle X0X01\rangle, \langle XXX01\rangle) = \langle XXX01\rangle \\ \chi^3 &= glb(\langle X0X01\rangle, \tau^{d(A_3)}_{A_3}(\chi^2)) = glb(\langle X0X01\rangle, \tau^{d(A_3)}_{A_3}(\langle XXX01\rangle)) \\ &= glb(\langle X0X01\rangle, \langle XXX01\rangle) = \langle XXX01\rangle \end{aligned}$$

and thus $\tilde{w} = Gfp\;\xi.\; glb(\langle X0X01\rangle, \tau^{d(A_3)}_{A_3}(\xi)) = \langle XXX01\rangle$. This computation shows that as long as clock signal n_3 is held low, node n_4 will retain its stored value of 0, and n_5 will remain at 1.

It is easy to verify that

$$\tau_{A_3}(\tilde{w}) = \langle XX001\rangle\langle XXX01\rangle\langle XXXX1\rangle\langle XXXXX\rangle\langle XXXXX\rangle\ldots$$

Since $\delta_{C_3} = \langle XXXX1\rangle\bot\bot\ldots$ it thus follows that $SAT(\tilde{w},\, [A_3 \Longrightarrow C_3])$. Finally, it follows trivially that $SAT(\tilde{w},\, [\text{true} \Longrightarrow \text{true}])$. Altogether, we can conclude that $SAT(\bot,\, G_2)$ holds.

We now return to the general theory by characterizing the satisfaction function. First we establish the following monotonicity property.

Proposition 3 *Given a trajectory assertion G, if $s \sqsubseteq t$ and $SAT(s,\, G)$ then $SAT(t,\, G)$.*

Proof: We prove the claim by induction on the structure of G. For the basis, $G = [A \Longrightarrow C]$, we have that $SAT(s,\, [A \Longrightarrow C])$ implies that $\delta_C \sqsubseteq \tau_A(s)$. However, by Lemma 3, it follows that $\tau_A(s) \sqsubseteq \tau_A(t)$ and thus $\delta_C \sqsubseteq \tau_A(t)$, which implies that $SAT(t,\, [A \Longrightarrow C])$. Now assume inductively that the claim holds for s, t and trajectory assertions $[A \Longrightarrow C]$ and G_1. If $G = [A \Longrightarrow C]\,;G_1$ then $SAT(s,\, G)$ implies that $SAT(s,\, [A \Longrightarrow C])$ and $SAT(\tau_A^{d(A)}(s),\, G_1)$. By the induction hypothesis it follows that $SAT(t,\, [A \Longrightarrow C])$. Furthermore, by Lemma 3 it follows that $\tau_A^{d(A)}(s) \sqsubseteq \tau_A^{d(A)}(t)$. This, together with the induction hypothesis, implies that $SAT(\tau_A^{d(A)}(t),\, G_1)$ and the claim follows. Finally, if $G = [A \Longrightarrow C]^*\,;G_1$ then $SAT(s,\, G)$ implies that $SAT(\tilde{s},\, [A \Longrightarrow C])$ and $SAT(\tilde{s},\, G_1)$ for $\tilde{s} = \mathit{Gfp}\ \xi.\ \mathit{glb}(s, \tau_A^{d(A)}(\xi))$. It follows directly from the definition of greatest fixed point that $\mathit{Gfp}\ \xi.\ \mathit{glb}(s, \tau_A^{d(A)}(\xi)) \sqsubseteq \mathit{Gfp}\ \xi.\ \mathit{glb}(t, \tau_A^{d(A)}(\xi)) = \tilde{t}$. Hence, by the induction hypothesis it follows that $SAT(\tilde{t},\, [A \Longrightarrow C])$ and $SAT(\tilde{t},\, G_1)$ and therefore that $SAT(t,\, G)$ and the induction step goes through and the claim follows. □

The following theorem constitutes one of the corner-stones in our verification methodology.

Theorem 1 *If G is an iteration-free trajectory assertion then for every $z \in \mathcal{S}$ we have*

$$L(\mathcal{M}, z) \models_{\mathcal{M}} G \quad \textit{iff} \quad SAT(z,\, G).$$

Proof: We prove the claim by induction over the structure of G. For the basis case, $G = [A \Longrightarrow C]$, we first show that if $\sigma \models_{\mathcal{M}} A$ implies that $\sigma \models_{\mathcal{M}} C$ for every $\sigma \in L(\mathcal{M}, z)$ then $\delta_C \sqsubseteq \tau_A(z)$. To establish this, let $\sigma = \tau_A(z)$. By Lemma 4 we know that $\tau_A(z) \in L(\mathcal{M}, z)$, and that $\tau_A(z) \models_{\mathcal{M}} A$. Hence, by assumption, $\tau_A(z) \models_{\mathcal{M}} C$. However, by Lemma 2 it follows that $\tau_A(z) \models_{\mathcal{M}} C$ iff $\delta_C \sqsubseteq \tau_A(z)$. Together, $\delta_C \sqsubseteq \tau_A(z)$.

To prove the converse, assume $\delta_C \sqsubseteq \tau_A(z)$. Consider an arbitrary $\sigma \in L(\mathcal{M}, z)$. There are two cases to consider: If $\tau_A(z) \not\sqsubseteq \sigma$ then by Lemma 4 it follows that $\sigma \not\models A$, and the claim follows. Hence, assume $\tau_A(z) \sqsubseteq \sigma$. This, together with our assumption that $\delta_C \sqsubseteq \tau_A(z)$, implies that $\delta_C \sqsubseteq \sigma$. Since $\sigma \in L(\mathcal{M}, z)$, Lemma 2 applies, and thus $\sigma \models_{\mathcal{M}} C$.

Now assume inductively that for any $x \in \mathcal{S}$ we have $L(\mathcal{M}, x) \models_{\mathcal{M}} G_1$ iff $SAT(x,\, G_1)$ and that $L(\mathcal{M}, x) \models_{\mathcal{M}} [A \Longrightarrow C]$ iff $SAT(x,\, [A \Longrightarrow C])$. If $G = [A \Longrightarrow C]\,;G_1$ then, by the truth semantics of G and Proposition 2, we have $L(\mathcal{M}, z) \models_{\mathcal{M}} G$ iff $L(\mathcal{M}, z) \models_{\mathcal{M}} [A \Longrightarrow C]$ and $L(\mathcal{M}, \tau_A^{d(A)}(z)) \models_{\mathcal{M}} G_1$. Together with the induction hypothesis we get that $L(\mathcal{M}, z) \models_{\mathcal{M}} G$ iff $SAT(z,\, [A \Longrightarrow C])$ and $SAT(\tau_A^{d(A)}(z),\, G_1)$. However, the latter holds iff $SAT(z,\, G)$. Consequently the induction step goes through and the claim follows. □

Our next theorem is the second major result of this section and provides the basis for our verification methodology. It shows that one direction of the claim made in Theorem 1 for iteration-free formulas also holds for general formulas. However, our fixed-point method for verifying formulas with iteration can cause overly pessimistic results, and therefore the other direction may not hold.

Theorem 2 *Let G be a trajectory assertion and let $z \in \mathcal{S}$. If $SAT(z, G)$ then $L(\mathcal{M}, z) \models_{\mathcal{M}} G$.*

Proof: We prove the result by induction on the structure of G. For the basis, if $G = \left[A^{[i]} \Longrightarrow C^{[i]}\right]$, for some $i \geq 1$, the claim follows immediately from Theorem 1. Now assume inductively that for any $x \in \mathcal{S}$, $SAT(x, \left[A^{[i]} \Longrightarrow C^{[i]}\right])$ implies that $L(\mathcal{M}, x) \models_{\mathcal{M}} \left[A^{[i]} \Longrightarrow C^{[i]}\right]$ for $i \geq 1$ and that $SAT(x, G_1)$ implies that $L(\mathcal{M}, x) \models_{\mathcal{M}} G_1$.

If $G = [A \Longrightarrow C]\,;G_1$ then $SAT(z, G)$ implies that $SAT(z, [A \Longrightarrow C])$ and $SAT(\tau_A^{d(A)}(z), G_1)$. By the induction hypothesis this implies that $L(\mathcal{M}, z) \models_{\mathcal{M}} [A \Longrightarrow C]$ and $L(\mathcal{M}, \tau_A^{d(A)}(z)) \models_{\mathcal{M}} G_1$, which together with Proposition 2 implies that $L(\mathcal{M}, z) \models_{\mathcal{M}} G$.

If $G = [A \Longrightarrow C]^*\,;G_1$ then, by Proposition 2, it follows that $L(\mathcal{M}, z) \models_{\mathcal{M}} [A \Longrightarrow C]^*\,;G_1$ iff $L(\mathcal{M}, z) \models_{\mathcal{M}} G_1$ and for $i \geq 1$, $L(\mathcal{M}, z) \models_{\mathcal{M}} \left[A^{[i]} \Longrightarrow C^{[i]}\right]$ and $L(\mathcal{M}, \tau_{A^{[i]}}^{d(A^{[i]})}(z)) \models_{\mathcal{M}} G_1$. Thus, in order to establish the induction step and show that $SAT(z, [A \Longrightarrow C]^*\,;G_1)$ implies $L(\mathcal{M}, z) \models_{\mathcal{M}} [A \Longrightarrow C]^*\,;G_1$ it suffices to prove that:

1. $SAT(z, [A \Longrightarrow C]^*\,;G_1)$ implies $L(\mathcal{M}, z) \models_{\mathcal{M}} G_1$,
2. $SAT(z, [A \Longrightarrow C]^*\,;G_1)$ implies $L(\mathcal{M}, z) \models_{\mathcal{M}} \left[A^{[i]} \Longrightarrow C^{[i]}\right]$ for $i \geq 1$, and
3. $SAT(z, [A \Longrightarrow C]^*\,;G_1)$ implies $L(\mathcal{M}, \tau_{A^{[i]}}^{d(A^{[i]})}(z)) \models_{\mathcal{M}} G_1$ for $i \geq 1$.

Before we prove the three cases, the following observations are useful. First, note that, by the definition of SAT, we have that $SAT(z, [A \Longrightarrow C]^*\,;G_1)$ implies that $SAT(\tilde{z}, G_1)$ and $SAT(\tilde{z}, [A \Longrightarrow C])$, where $\tilde{z} = \mathit{Gfp}\ \xi.\ \mathit{glb}(z, \tau_A^{d(A)}(\xi))$. Also, it is easy to verify that, by the definition of *glb* and the definition of the fixed point equation, we have $\tilde{z} \sqsubseteq z$ and $\tilde{z} \sqsubseteq \tau_{A^{[i]}}^{d(A^{[i]})}(z)$ for $i \geq 1$.

By Proposition 3 and the fact that $\tilde{z} \sqsubseteq z$ it follows that if $SAT(\tilde{z}, G_1)$ then $SAT(z, G_1)$. This, together with the induction hypothesis implies that $L(\mathcal{M}, z) \models_{\mathcal{M}} G_1$ and the first claim is established.

To prove that $L(\mathcal{M}, z) \models_{\mathcal{M}} \left[A^{[i]} \Longrightarrow C^{[i]}\right]$ for any $i \geq 1$, we first note that, by definition, $SAT(\tilde{z}, [A \Longrightarrow C])$ holds iff $\delta_C \sqsubseteq \tau_A(\tilde{z})$. We will now prove, by induction on i, that if $\delta_C \sqsubseteq \tau_A(\tilde{z})$ then $\delta_{C^{[i]}} \sqsubseteq \tau_{A^{[i]}}(z)$. Given this result the second claim follows trivially from the definition of SAT and the induction hypothesis. For the basis, $i = 1$, note that $A^{[1]} = A$ and $C^{[1]} = C$. Furthermore, since $\tilde{z} \sqsubseteq z$ and thus, by Lemma 3, we can infer that if $SAT(\tilde{z}, [A \Longrightarrow C])$ then $SAT(z, [A \Longrightarrow C])$. Altogether, we can conclude that $\delta_{C^{[1]}} \sqsubseteq \tau_{A^{[1]}}(z)$. Now assume inductively that $SAT(\tilde{z}, [A \Longrightarrow C])$ implies $\delta_{C^{[i]}} \sqsubseteq \tau_{A^{[i]}}(z)$ for some $i \geq 1$ and consider $i + 1$. By the definition of $\delta_{C^{[i+1]}}$ and $\tau_{A^{[i+1]}}(z)$ and the assumption that $d(A) = d(C)$ we have $\delta^j_{C^{[i+1]}} = \delta^j_{C^{[i]}}$ and $\tau^j_{A^{[i+1]}}(z) = \tau^j_{A^{[i]}}(z)$ for $0 \leq j < d(A^{[i]})$. Thus in order to show that $\delta_{C^{[i+1]}} \sqsubseteq \tau_{A^{[i+1]}}(z)$ we only need to show that $\mathit{suffix}(d(A^{[i]}), \delta_{C^{[i+1]}}) \sqsubseteq \mathit{suffix}(d(A^{[i]}), \tau_{A^{[i+1]}}(z))$. However, from the definition of $\delta_{C^{[i+1]}}$ and $\tau_{A^{[i+1]}}(z)$ it follows that $\mathit{suffix}(d(A^{[i]}), \delta_{C^{[i+1]}}) = \delta_C$ and $\mathit{suffix}(d(A^{[i]}), \tau_{A^{[i+1]}}(z)) = \tau_A(\tau_{A^{[i]}}^{d(A^{[i]})}(z))$. As above $\tilde{z} \sqsubseteq \tau_{A^{[i]}}^{d(A^{[i]})}(z)$ and thus, by Lemma 3, it follows that:

$$SAT(\tilde{z}, [A \Longrightarrow C]) \quad \text{implies} \quad SAT(\tau_{A^{[i]}}^{d(A^{[i]})}(z), [A \Longrightarrow C]).$$

In other words, $\delta_C \sqsubseteq \tau_A(\tau_{A^{[i]}}^{d(A^{[i]})}(z))$ and therefore $\delta_{C^{[i+1]}} \sqsubseteq \tau_{A^{[i+1]}}(z)$. Altogether, if $SAT(\tilde{z}, [A \Longrightarrow C])$ then $\delta_{C^{[i+1]}} \sqsubseteq \tau_{A^{[i+1]}}(z)$ and the induction step goes through and the claim follows.

Finally, since $SAT(\tilde{z}, G_1)$ and $\tilde{z} \sqsubseteq \tau_{A^{[i]}}^{d(A^{[i]})}(z)$ for $i \geq 1$, it follows directly from Lemma 3, that $SAT(\tau_{A^{[i]}}^{d(A^{[i]})}(z), G_1)$. This, together with the induction hypothesis implies that $L(\mathcal{M}, \tau_{A^{[i]}}^{d(A^{[i]})}(z)) \models_{\mathcal{M}}$

G_1 and the third claim follows. □

The way we are representing sets of states during the fixed point calculation by the greatest lower bound of the states in the set has some undesirable properties. In particular, if the lattice is "too sparse", so that a very general state must be used to represent a set of states, it is quite likely that we will lose too much information and thus may find that SAT does not hold even though a more accurate calculation would show that the trajectory assertion is valid. Of course, from the above theorems we know that this can only happen if we have iterations in the trajectory assertion.

To illustrate the problem of too sparse lattices, assume we have a circuit that contains a "sticky" 2-bit wait-state counter that sequences through the states $\langle 00\rangle$, $\langle 01\rangle$, and $\langle 10\rangle$, but no further, no matter how many input pulses it receives. Suppose we want to check this counter by using an iteration construct. If we first use the standard switch-level lattice introduced in Section 3, it is easy to see that the fixed point calculation will be forced to set both nodes of the counter to X since $\langle XX\rangle = glb\{\langle 00\rangle, \langle 01\rangle, \langle 10\rangle\}$. Unfortunately, we have now lost information and thus we may erroneously report a circuit failure that only could be triggered if the counter ended up in the state $\langle 11\rangle$. On the other hand, if we used a more complete lattice the problem would disappear. For example, if we use the power-set of $\{\langle 00\rangle, \langle 01\rangle, \langle 10\rangle, \langle 11\rangle\}$ ordered by set inclusion as the domain of the counter, we can distinguish between the set $\{\langle 00\rangle, \langle 01\rangle, \langle 10\rangle\}$ and any set that contains the state $\langle 11\rangle$. As an alternative to increasing the computational complexity through more detailed lattice structures, we can work around the limited power of our iteration construct by specifying the explicit circuit behavior until it stabilizes. For example, we would explicitly check that the counter goes through states $\langle 00\rangle$ and $\langle 01\rangle$, and then use an iteration construct to check that the counter remains in state $\langle 10\rangle$ beyond this point.

The above theorem suggests a simple method for verifying a trajectory assertion G: compute $SAT(\perp, G)$. If G is iteration-free then we will obtain an exact answer in the sense that $SAT(\perp, G)$ holds if and only if $\models_{\mathcal{M}} G$ holds. On the other hand, if there are iterations in G, then we can only guarantee that if $SAT(\perp, G)$ then $\models_{\mathcal{M}} G$. Unfortunately, there is a practical difficulty with this approach since all the defining trajectories and the defining sequences are, as defined in the previous section, infinite. Note, however, that the fixed point calculation does not require us to compute an infinite defining trajectory since we only need to compute $\tau_A(\xi)$ for various ξ up to $d(A)$. Also, by Corollary 1, in order to compare a defining sequence with a defining trajectory in computing the satisfaction function, it is sufficient to compute a bounded prefix of the defining trajectories and the defining sequences. Hence, we only need to compute a bounded prefix of any trajectory. Furthermore, it is easy to see that we never need to store more than three system states: the current state, the next state, and the fixed point state if the assertion contains an iteration. In summary, we can verify trajectory assertions very efficiently.

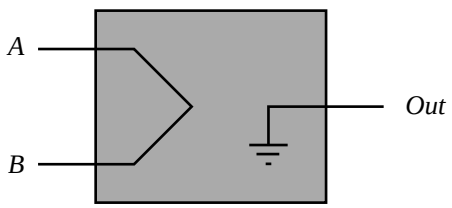

Figure 6: Pseudo XOR circuit.

Finally, there is one more, quite subtle, aspect of the verification methodology we need to deal

with. The problem is that in order to make a non-lattice domain into a complete lattice, we often add "artificial" top elements. Since every element is less than the top element, we are in a somewhat dangerous situation if, during the computation of the defining trajectory, we end up in such a top state. To illustrate a typical instance of this problem, 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 satisfies the assertion $[(\texttt{A is } a) \wedge (\texttt{B is } b) \Longrightarrow \mathbf{N}(\texttt{Out is } a \oplus b)]$, for all $a, b \in \{0, 1\}$. Unfortunately, this is not the case. For example, this assertion is satisfied by the rather useless circuit of Fig. 6, where the two inputs are tied together, and the output is always 0. Whenever $a \neq b$ the antecedent trajectory will end up in $\top$, because inputs `A` and `B` are electrically equivalent. The only values for which the trajectory does not end up in $\top$ 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 implies everything" error. Problems of this sort have been encountered by researchers using other systems for hardware verification such as HOL [24] and EMC [16]. A solution to this problem in our context, and in fact the solution we have adapted for our prototype tools, is a two-pronged approach. First, the user can only add new top elements in forming a complete lattice. Thus we do not allow the user to add artificial bottom or internal states. Secondly, our verification system ensures that every state in the defining trajectory does not contain any artificially introduced top elements. These two constraints ensure that the defining trajectory is a genuine circuit trajectory, and thus there is at least one circuit trajectory satisfying the antecedent.

7 Symbolic Formulation

In the previous section we proved that to determine the validity of a trajectory assertion G it suffices to compute $SAT(\bot, G)$. Unfortunately, when verifying all but a limited class of systems (including many memory designs [11]) we would need to write down and verify an exponentially large number of assertions. The coverage of multiple cases by the partially-ordered system model lacks sufficient precision to reliably verify the many distinct operating conditions.

In this section we first extend the trajectory formulas by introducing symbolic trajectory formulas. Each symbolic trajectory formula can express a large number of assertions that the behavior of the system must obey. We then introduce a method of verifying such a collection of 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. In essence, we will perform the case analysis implicitly rather than explicitly.

7.1 Symbolic Expressions

Let $\mathcal{V}$ be a set of symbolic Boolean variables. For convenience, let $\mathcal{B}$ denote the set $\{0, 1\}$. An *assignment*, ϕ, is a mapping $\phi: \mathcal{V} \rightarrow \mathcal{B}$ assigning a binary value to each variable. Let Φ be the set of all possible assignments, i.e., $\Phi = \{\phi: \mathcal{V} \rightarrow \mathcal{B}\}$. 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 mapping $e: \Phi \rightarrow \mathcal{B}$ and thus denotes the domain constraint $\mathcal{D} = \{\phi \mid e(\phi) = 1\}$. The set of all assignments Φ is denoted by the constant function $\dot{1}$, defined as yielding 1 for all assignments. Expressing domain constraints by Boolean expressions allows us to compactly specify many different circuit operating conditions with a single formula.

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

In general, if $\mathcal{D}$ is a scalar domain set we extend it to a symbolic domain set, written $\mathcal{D}(\mathcal{V})$, by defining

$$\mathcal{D}(\mathcal{V}) = \{f: \Phi \to \mathcal{D}\}.$$

In other words, $\mathcal{D}(\mathcal{V})$ denotes the set of functions mapping an assignment in Φ to $\mathcal{D}$.

For any element a of $\mathcal{D}$, we let $\dot{a}$ denote the *constant* function, yielding $\dot{a}(\phi) = a$ for any assignment ϕ.

We extend all operations from scalar to symbolic domains in a uniform way. Consider an operation $op: \mathcal{D}_1 \times \mathcal{D}_2 \to \mathcal{D}_3$, defined over scalar domains $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$. Its symbolic counterpart $\dot{op}: \mathcal{D}_1(\mathcal{V}) \times \mathcal{D}_2(\mathcal{V}) \to \mathcal{D}_3(\mathcal{V})$ is defined such that for all $\dot{a} \in \mathcal{D}_1(\mathcal{V})$ and $\dot{b} \in \mathcal{D}_2(\mathcal{V})$, we have $(\dot{a}\ \dot{op}\ \dot{b})(\phi) = \dot{a}(\phi)\ op\ \dot{b}(\phi)$.

When extending a relation R symbolically, we define the result to be a function specifying the assignments under which its arguments are related. In other words, we actually extend the characteristic function of the relation. That is, given a binary relation $R \subseteq \mathcal{D}_1 \times \mathcal{D}_2$, define $\dot{R}: \mathcal{D}_1(\mathcal{V}) \times \mathcal{D}_2(\mathcal{V}) \to \mathcal{B}(\mathcal{V})$ as $(\dot{a}\ \dot{R}\ \dot{b})(\phi) = 1$ if and only if $\dot{a}(\phi)\ R\ \dot{b}(\phi)$.

7.2 Symbolic Trajectory Formulas and Assertions

A (scalar) trajectory formula expresses a constraint on a trajectory. We now extend this idea by introducing symbolic trajectory formulas. A symbolic trajectory formula expresses a set of constraints on a trajectory by representing a set of (scalar) trajectory formulas. More specifically, a symbolic trajectory formula will be a function mapping an assignment $\phi \in \Phi$ to a trajectory formula.

Trajectory formulas can be extended to symbolic trajectory formulas in several ways. We will present one particular definition here that is intuitively simple, yet powerful enough to make specifications of desirable system properties fairly natural.

Assume $\langle \mathcal{S}, \sqsubseteq \rangle$ is a lattice, $\mathcal{V}$ is a set of symbolic Boolean variables, and $\mathcal{P}$ is a set of simple predicates over $\mathcal{S}$. A *symbolic trajectory formula* is defined recursively as:

1. **Simple predicates**: p is a symbolic trajectory formula if $p \in \mathcal{P}$.
2. **Conjunction**: $(\dot{F}_1 \wedge \dot{F}_2)$ is a symbolic trajectory formula if $\dot{F}_1$ and $\dot{F}_2$ are symbolic trajectory formulas.
3. **Domain restriction**: $(E \to \dot{F})$ is a symbolic trajectory formula if $\dot{F}$ is a symbolic trajectory formula and E is a Boolean expression over $\mathcal{V}$
4. **Next time**: $(\mathbf{N}\dot{F})$ is a symbolic trajectory formula if $\dot{F}$ is a symbolic trajectory formula.

Note that the only change from the definition of trajectory formulas is that the domain constraint can now be a Boolean expression rather than only 1 or 0.

For the case of switch-level circuits, we introduce the notation $(n_i\ \mathbf{is}\ E)$ as a shorthand for the formula $(E \to (n_i\ \mathbf{is}\ 1)) \wedge (\overline{E} \to (n_i\ \mathbf{is}\ 0))$. That is, we constrain node n_i to have the particular symbolic Boolean value denoted by the expression E.

The concept of depth is extended to the symbolic domain in the natural way, i.e., the depth of a symbolic trajectory formula is one greater than the number of nested next time operators.

A *symbolic trajectory assertion* is defined recursively as:

1. **Simple assertions:** $\left[\dot{A} \Longrightarrow \dot{C}\right]$, where $\dot{A}$ and $\dot{C}$ are symbolic trajectory formulas and $d(\dot{A}) = d(\dot{C})$.

2. **Sequences:** $\left[\dot{A} \Longrightarrow \dot{C}\right]; \dot{G}_1$, where $\dot{A}$ and $\dot{C}$ are symbolic trajectory formulas, $d(\dot{A}) = d(\dot{C})$, and $\dot{G}_1$ is a symbolic trajectory assertion.

3. **Iterations:** $\left[\dot{A} \Longrightarrow \dot{C}\right]^*; \dot{G}_1$, where $\dot{A}$ and $\dot{C}$ are symbolic trajectory formulas, $d(\dot{A}) = d(\dot{C})$, and $\dot{G}_1$ is a symbolic trajectory assertion.

With the above development, including our shorthand notation, we can now combine our two trajectory assertions that constitute our specification of the unit-delay inverter circuit of Fig. 2 into one symbolic trajectory assertion as follows. Assume $\mathcal{V} = \{x\}$, then

$$[(\texttt{in is } x) \wedge \mathbf{N}\mathit{true} \Longrightarrow \mathbf{N}(\texttt{out is } \overline{x})].$$

As a more complex example, consider the following symbolic trajectory assertion for the latch circuit of Fig. 5. Here, assume that $\mathcal{V} = \{c, a\}$. We have the symbolic assertion

$$G_3 = \left[(n_3 \textbf{ is } c)^{[2]} \wedge (c \rightarrow (n_1 \textbf{ is } a))^{[2]} \wedge (\overline{c} \rightarrow (n_4 \textbf{ is } \overline{a})) \Longrightarrow \mathbf{N}^2(n_4 \textbf{ is } \overline{a})\right].$$

Informally, the antecedent states that depending on the c ("clock") variable we either load value a into the latch (by setting n_3 to 1 and n_1 to a) or we assume that a is already stored in the latch (with n_3 set to 0 and n_4 to $\overline{a}$). The consequent states that value a is stored in the latch on the third time unit.

Given a symbolic trajectory formula $\dot{F}$ and an assignment $\phi \in \Phi$, the corresponding trajectory formula, written $\dot{F}(\phi)$, is defined recursively as:

1. $p(\phi) \stackrel{\text{def}}{=} p$ if $p \in \mathcal{P}$.

2. $(\dot{F}_1 \wedge \dot{F}_2)(\phi) \stackrel{\text{def}}{=} (\dot{F}_1(\phi) \wedge \dot{F}_2(\phi))$.

3. $(E \rightarrow \dot{F})(\phi) \stackrel{\text{def}}{=} (e(\phi) \rightarrow \dot{F}(\phi))$, where e is the Boolean function denoted by E.

4. $(\mathbf{N}\dot{F})(\phi) \stackrel{\text{def}}{=} (\mathbf{N}(\dot{F}(\phi)))$.

Similarly, given a symbolic trajectory assertion $\dot{G}$ and an assignment $\phi \in \Phi$, the corresponding trajectory assertion, written $\dot{G}(\phi)$, is defined recursively as:

1. $\left[\dot{A} \Longrightarrow \dot{C}\right](\phi) \stackrel{\text{def}}{=} \left[\dot{A}(\phi) \Longrightarrow \dot{C}(\phi)\right]$.

2. $(\left[\dot{A} \Longrightarrow \dot{C}\right]; \dot{G}_1)(\phi) \stackrel{\text{def}}{=} \left[\dot{A}(\phi) \Longrightarrow \dot{C}(\phi)\right]; (\dot{G}_1(\phi))$.

3. $(\left[\dot{A} \Longrightarrow \dot{C}\right]^*; \dot{G}_1)(\phi) \stackrel{\text{def}}{=} \left[\dot{A}(\phi) \Longrightarrow \dot{C}(\phi)\right]^*; (\dot{G}_1(\phi))$.

Given the above, we can now extend the $\models_{\mathcal{M}}$ relation to the symbolic domain in the standard way, i.e., if $\dot{F}$ is a symbolic trajectory formula then for every $\sigma \in L(\mathcal{M})$ we have

$$(\sigma \dot{\models}_{\mathcal{M}} \dot{F})(\phi) = 1 \quad \text{iff} \quad \sigma \models_{\mathcal{M}} (\dot{F}(\phi)).$$

Similarly, if $\dot{G}$ is a symbolic trajectory assertion then for any set L of trajectories we have

$$(L \dot{\models}_{\mathcal{M}} \dot{G})(\phi) = 1 \quad \text{iff} \quad L \models_{\mathcal{M}} (\dot{G}(\phi)).$$

Now, given a model structure $\mathcal{M}$ and a symbolic assertion $\dot{G}$, the task of our checking algorithm is to compute the Boolean function expressing the set of assignments 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 assignments.

7.3 Checking Symbolic Trajectory Assertions

In Section 5, we showed how scalar trajectory assertions can be verified very efficiently by computing the satisfaction predicate. By extending the functions and relations used in this process to the symbolic domain, we can perform the same algebraic manipulations. Rather than a true/false answer, we obtain a Boolean function denoting those assignments ϕ for which the assertion holds.

Define the symbolic domains $\mathcal{B}(\mathcal{V})$, $\mathcal{S}(\mathcal{V})$, and $\mathcal{S}^\omega(\mathcal{V})$ as denoting the set of functions mapping an assignment in Φ to $\mathcal{B}$, $\mathcal{S}$, and $\mathcal{S}^\omega$ respectively. Let $\dot{Y}$, $\dot{lub}$, $\dot{glb}$, $\dot{Gfp}$, and $\dot{?}$ denote the symbolic extensions of the successor function Y, the *lub* operation, the *glb* operation, the *Gfp* operation, and the infix ? operation respectively. Let $\dot{\sqsubseteq}$ be the extension of the ordering relation $\sqsubseteq$ to the symbolic domain. Recall that a relation over a scalar domain extends symbolically to a function specifying the assignments under which its arguments are related. The normal Boolean product operation $\cdot$ serves as the symbolic extension of the logical "and" connective. That is, for any assignment ϕ $(\dot{a} \cdot \dot{b})(\phi) = 1$ iff $\dot{a}(\phi) = 1$ and $\dot{b}(\phi) = 1$. Recall that for a state value $a \in \mathcal{S}$, $\dot{a}$ denotes the constant function, yielding a for all assignments. In particular, $\dot{\perp}$ denotes the function that always yields $\perp$.

Given a symbolic trajectory formula $\dot{F}$, we define its *defining symbolic sequence* $\dot{\delta}_{\dot{F}}$ recursively as follows:

1. $\dot{\delta}_p = \dot{\overline{p}}\dot{\perp}\dot{\perp}\ldots$ if $p \in \mathcal{P}$ is a simple predicate with defining value $\overline{p}$.
2. $\dot{\delta}_{\dot{F}_1 \wedge \dot{F}_2} = \dot{lub}(\dot{\delta}_{\dot{F}_1}, \dot{\delta}_{\dot{F}_2})$.
3. $\dot{\delta}_{E \rightarrow \dot{F}} = e\dot{?}\dot{\delta}_{\dot{F}}$, where e is the Boolean function denoted by E.
4. $\dot{\delta}_{\mathbf{N}\dot{F}} = \dot{\perp}\dot{\delta}_{\dot{F}}$.

Proposition 4 *Let $\dot{F}$ be a symbolic trajectory formula and let $\dot{\delta}_{\dot{F}}$ be its defining symbolic sequence. Then, $(\dot{\delta}_{\dot{F}})(\phi) = \delta_{\dot{F}(\phi)}$, for every $\phi \in \Phi$.*

Proof: Follows directly from the definition of symbolic trajectory formulas and the definitions of $\mathcal{S}^\omega$, $\dot{lub}$ and $\dot{?}$. □

Given a symbolic starting state $\dot{z} \in \mathcal{S}(\mathcal{V})$ and symbolic trajectory formula $\dot{F}$ with defining symbolic sequence $\dot{\delta}_{\dot{F}} = \dot{\delta}^0_{\dot{F}}\dot{\delta}^1_{\dot{F}}\ldots$, the *defining symbolic trajectory* $\dot{\tau}_{\dot{F}}(\dot{z}) = \dot{\tau}^0_{\dot{F}}(\dot{z})\dot{\tau}^1_{\dot{F}}(\dot{z})\ldots$ is defined inductively as follows:

$$\dot{\tau}^i_{\dot{F}}(\dot{z}) = \begin{cases} \dot{lub}(\dot{\delta}^0_{\dot{F}}, \dot{z}) & \text{if } i = 0 \\ \dot{lub}(\dot{\delta}^i_{\dot{F}}, \dot{Y}(\dot{\tau}^{i-1}_{\dot{F}}(\dot{z}))) & \text{otherwise} \end{cases}$$

Proposition 5 *If $\dot{F}$ is a symbolic trajectory formula and $\dot{z} \in \mathcal{S}(\mathcal{V})$ let $\dot{\tau}_{\dot{F}}(\dot{z})$ be the defining symbolic trajectory for F. Then $(\dot{\tau}_{\dot{F}}(\dot{z}))(\phi) = \tau_{\dot{F}(\phi)}(\dot{z}(\phi))$ for every $\phi \in \Phi$.*

Proof: Follows directly from the definition of symbolic trajectory formulas, Proposition 4, and the definitions of $\dot{Y}$, and $\dot{lub}$. □

Now, given a symbolic trajectory assertion $\dot{G}$ define its symbolic satisfaction predicate $\dot{SAT}$ as follows:

1. $\dot{SAT}(\dot{z}, \left[\dot{A} \Longrightarrow \dot{C}\right]) = (\dot{\delta}_{\dot{C}} \dot{\sqsubseteq} \dot{\tau}_{\dot{A}}(\dot{z}))$.

2. $\dot{SAT}(\dot{z},\ \left[\dot{A} \Longrightarrow \dot{C}\right];\dot{G}_1) = (\dot{SAT}(\dot{z},\ \left[\dot{A} \Longrightarrow \dot{C}\right]) \cdot \dot{SAT}(\dot{\tau}_{\dot{A}}^{d(\dot{A})}(\dot{z}),\ \dot{G}_1))$.

3. $\dot{SAT}(\dot{z},\ \left[\dot{A} \Longrightarrow \dot{C}\right]^*;\dot{G}_1) = (\dot{SAT}(\tilde{\dot{z}},\ \dot{G}_1) \cdot \dot{SAT}(\tilde{\dot{z}},\ \left[\dot{A} \Longrightarrow \dot{C}\right]))$, where

$$\tilde{\dot{z}} = \dot{Gfp}\ \dot{\xi}.\ \dot{glb}(\dot{z}, \dot{\tau}_{\dot{A}}^{d(\dot{A})}(\dot{\xi})).$$

In view of the above results and Theorems 1 and 2 the following theorem follows immediately.

Theorem 3 *Assume $\dot{G}$ is a symbolic trajectory assertion. Then for every $\phi \in \Phi$:*

$$\dot{SAT}(\dot{\perp},\ \dot{G})(\phi) = 1 \quad \textit{implies} \quad (\dot{\models}_{\mathcal{M}}\dot{G})(\phi) = 1,$$

Furthermore, if $\dot{G}$ is iteration-free, then

$$\dot{SAT}(\dot{\perp},\ \dot{G})(\phi) = 1 \quad \textit{iff} \quad (\dot{\models}_{\mathcal{M}}\dot{G})(\phi) = 1.$$

To illustrate the practical application of Theorem 3 consider the symbolic trajectory assertion G_3 defined as

$$G_3 = \left[(n_3 \textbf{ is } c)^{[2]} \wedge (c \rightarrow (n_1 \textbf{ is } a))^{[2]} \wedge (\overline{c} \rightarrow (n_4 \textbf{ is } \overline{a})) \Longrightarrow \mathbf{N}^2(n_4 \textbf{ is } \overline{a})\right].$$

Assume we want to check this formula for the model structure corresponding to the circuit of Fig. 5. We will show the computation of the symbolic defining sequence and the symbolic defining trajectory. In order to do so, however, we must introduce an expression syntax for symbolic ternary values, i.e., functions mapping Boolean assignments to ternary values. Following our earlier convention, we will let $\dot{X}$ denote the constant function for value X. We will use Boolean expressions to denote cases where all assignments yield binary node values. Finally, for Boolean expression E_t, and symbolic ternary expressions E_1, and E_0 we will use the notation $E_t \dot{\rightarrow} E_1 \mid E_0$ to denote the function

$$(e_t \dot{\rightarrow} e_1 \mid e_0)(\phi) = \begin{cases} e_1(\phi) & \text{if } e_t(\phi) = 1 \\ e_0(\phi) & \text{otherwise,} \end{cases}$$

where e_t, e_1, and e_0 are the functions denoted by the expressions E_t, E_1, and E_0, respectively.

First, for the antecedent $A = (n_3 \textbf{ is } c)^{[2]} \wedge (c \rightarrow (n_1 \textbf{ is } a))^{[2]} \wedge (\overline{c} \rightarrow (n_4 \textbf{ is } \overline{a}))$, we obtain the following elements for the defining sequence and trajectory:

i	$\dot{\delta}_A^i$					$\dot{Y}(\dot{\tau}_A^{i-1}(\dot{\perp}))$					$\dot{\tau}_A^i(\dot{\perp})$				
	n_1	n_2	n_3	n_4	n_5	n_1	n_2	n_3	n_4	n_5	n_1	n_2	n_3	n_4	n_5
0	$c\dot{\rightarrow}a \mid \dot{X}$	$\dot{X}$	c	$c\dot{\rightarrow}\dot{X} \mid \overline{a}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$c\dot{\rightarrow}a \mid \dot{X}$	$\dot{X}$	c	$c\dot{\rightarrow}\dot{X} \mid \overline{a}$	$\dot{X}$
1	$c\dot{\rightarrow}a \mid \dot{X}$	$\dot{X}$	c	$\dot{X}$	$\dot{X}$	$\dot{X}$	$c\dot{\rightarrow}\overline{a} \mid \dot{X}$	$\dot{X}$	$\overline{a}$	$c\dot{\rightarrow}\dot{X} \mid a$	$c\dot{\rightarrow}a \mid \dot{X}$	$c\dot{\rightarrow}\overline{a} \mid \dot{X}$	c	$\overline{a}$	$c\dot{\rightarrow}\dot{X} \mid a$
2	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$c\dot{\rightarrow}\overline{a} \mid \dot{X}$	$\dot{X}$	$\overline{a}$	a	$\dot{X}$	$c\dot{\rightarrow}\overline{a} \mid \dot{X}$	$\dot{X}$	$\overline{a}$	a
3	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	a	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	a
≥ 4	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$	$\dot{X}$

Similarly, it is easy to see that

$$\dot{\delta}_C = \langle \dot{X}\dot{X}\dot{X}\dot{X}\dot{X}\rangle\langle \dot{X}\dot{X}\dot{X}\dot{X}\dot{X}\rangle\langle \dot{X}\dot{X}\dot{X}\overline{a}\dot{X}\rangle\langle \dot{X}\dot{X}\dot{X}\dot{X}\dot{X}\rangle\langle \dot{X}\dot{X}\dot{X}\dot{X}\dot{X}\rangle \ldots$$

and thus that $\dot{\delta}_C \dot{\sqsubseteq} \dot{\tau}_A = \dot{1}$, i.e., the assertion holds for all variable assignments.

8 Extensions to the Logic

The base logic, as described above, is convenient for deriving the underlying theory. Unfortunately, expressing "interesting" assertions about real systems using only the constructs given in Section 4 is very tedious. Two shortcomings make using the logic cumbersome: the fine granularity of the timing, and the lack of more powerful logical constructs. We have already introduced several shorthand notations that take partial steps in remedying these limitations. In general, one can increase the expressive power of the logic greatly by introducing further shorthands. The semantics of each such extension is defined by a syntactic translation into the base logic, and hence has a well-defined semantics and implementation.

In order to define a language for writing specifications we need to define three entities: the syntax of the language, the semantics of the language, and a compilation algorithm that can translate the high-level constructs to the core logic. Furthermore, in order not to get astray in the process, a properly defined compiler function should also be proven correct in the sense that the semantics of the higher-level constructs are preserved by the compilation process. Although we will describe the extensions we have made in fairly informal terms, Joyce and Seger [27, 35] has in fact formalized a very similar language in higher-order logic and there proven that the compilation algorithm is correct. Also, as a side effect of properly formalizing the semantics of the added constructs, we open up the possibility of reasoning about the specifications themselves [35].

8.1 Timing Extensions

We have already introduced the notation $F^{[k]}$ to denote that property F should hold for k successive time intervals, where each interval has duration given by the depth $d(F)$. This concept can be generalized to other sequencing constructs such as *during*, *from–to*, *then*, and *for*. With these we can, for example, write $((p_1 \mathit{\ for\ } 100) \wedge (p_2 \mathit{\ for\ } 100))\ \mathit{then}\ (p_3 \mathit{\ for\ } 10)$ rather than having to write

$$(p_1 \wedge p_2) \wedge \mathbf{N}(p_1 \wedge p_2) \wedge \mathbf{N}^2(p_1 \wedge p_2) \wedge \ldots \wedge \mathbf{N}^{99}(p_1 \wedge p_2) \wedge \mathbf{N}^{100}p_3 \wedge \ldots \mathbf{N}^{109}p_3.$$

Each of these constructs has a straightforward definition in terms of our existing notation. As an illustration, the duration construct, written $\mathit{during}\ (s, e, F)$, has as arguments a start time s, an end time e and an instantaneous trajectory formula F that is to hold over this interval. This can be translated simply as *true* for $e < s$, or $\mathbf{N}^s F^{[e-s+1]}$ for $e \geq s$.

We have also seen that for most sequential circuits, reasoning at the unit step level is far too tedious. Instead, we would like to write and verify specifications at a more abstract timing level. For example, with *phase-level* timing, we view each period when the clocks are held at fixed values to be a phase, and assume that each phase has some minimum length k [7]. For simplicity, we will first assume that all phases have the same duration. A naive approach to phase-level timing would be to translate an instantaneous phase formula F into $F^{[k]}$, and introduce A "next phase" operator $\mathbf{N}_p$ defined simply as $\mathbf{N}^k$. That is, any property F should hold throughout the phase, and each successive phase starts exactly k time units from its predecessor.

Although the above attempt at phase-level timing frees us from describing the desired behaviors for every basic time unit, it has a serious drawback. The problem lies in the fact we must specify the precise length of the phase. As a result, we overspecify the desired behavior. In fact, we only show that the system works when all phases are exactly k basic time units long. Instead, we would like to verify that the system works correctly as long as each phase is *at least* k time units long. As was shown in Section 6 this can be accomplished by using the iteration construct of trajectory assertions.

To illustrate the problem with fixed length phases and how it can be remedied, consider the switch-level circuit of Fig. 7. Intuitively, n_1 is the (inverted) input to a latch, n_3 is the clock signal,

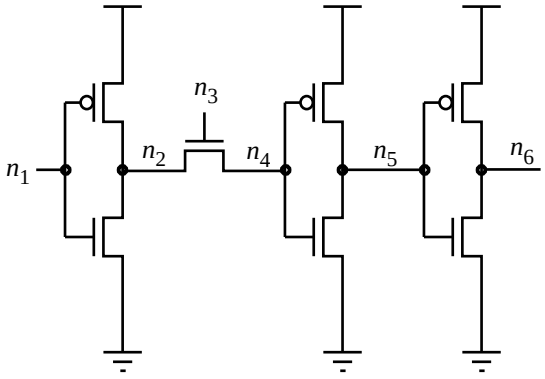

Figure 7: Circuit illustrating the use of iteration.

n_4 is the electrical node that stores the state when the clock is low, and n_6 is the output of the output buffer. Suppose we are trying to determine whether a 0 stored in the latch will remain to the end of the phase even if the clock goes high. Clearly, this is a property that a latch should not satisfy, but if we assume that each phase is exactly 2 time units long, we could arrive at this false conclusion. In order to check the validity of the statement by our naive model, the following assertion would be used:

$$\left[(n_4 \textbf{ is } 0) \wedge (n_3 \textbf{ is } 0)^{[2]} \Longrightarrow true^{[2]}\right]; \left[(n_3 \textbf{ is } 1)^{[2]} \Longrightarrow (n_6 \textbf{ is } 0)^{[2]}\right].$$

The circuit in Fig. 7 satisfies this assertion, because there is a 2 unit propagation delay from storage node n_4 to output n_6. If we assume the phases to be 3 time units, and thus we try to check the assertion

$$\left[(n_4 \textbf{ is } 0) \wedge (n_3 \textbf{ is } 0)^{[2]} \Longrightarrow true^{[3]}\right]; \left[(n_3 \textbf{ is } 1)^{[3]} \Longrightarrow (n_6 \textbf{ is } 0)^{[3]}\right],$$

it is easy to see that the circuit in Fig. 7 does not satisfy the assertion. In order to avoid this apparently "non-monotonic" behavior, it is preferable to check an assertion like:

$$\left[(n_4 \textbf{ is } 0) \wedge (n_3 \textbf{ is } 0)^{[2]} \Longrightarrow true^{[2]}\right]; [n_3 \textbf{ is } 0 \Longrightarrow true]^*;$$
$$\left[(n_3 \textbf{ is } 1)^{[2]} \Longrightarrow (n_6 \textbf{ is } 0)^{[2]}\right]; [(n_3 \textbf{ is } 1) \Longrightarrow (n_6 \textbf{ is } 0)]^* [true \Longrightarrow true].$$

where we have used the iteration construct to make sure the property we are checking holds no matter how long the phases are. It is easy to see that this assertion will fail for the circuit shown in Fig. 7. In particular, the last iteration assertion will fail.

We can generalize the above approach by defining a "stable phase assert" command. Assume we would like to check some assertion $[A \Longrightarrow C]$, where A and C are instantaneous formulas, during a phase. Assume furthermore that phases are at least k time units long. The "stable phase" assert command would be a shorthand for $\left[A^{[k]} \Longrightarrow C^{[k]}\right]; [A \Longrightarrow C]^*$. In essence, we allow the circuit to take k time units to reach a stable state. We then prove that $[A \Longrightarrow C]$ is an invariant of the system beyond these k units. The set of invariant conditions beyond the k time units is captured by a state which is less than or equal to every possible state of the system as long as A continues to hold. We would continue the verification of further properties from this state.

Interestingly, this phase-level timing implements a form of "oscillation control" that was included in the original COSMOS simulator [7]. In the simulator, the user specifies a limit on the

phase length k. When simulating a phase, the simulator computes new states for nodes until it reaches a stable state. Once the limit k on unit steps is taken, however, any node changing state is set to X rather than to its excitation. This procedure matches exactly the fixed-point implementation of the iteration construct for the ternary domain. In fact, our symbolic simulator implements the fixed-point approach in its full generality.

8.2 Data Handling Extensions

There are several extensions that simplify the task of writing specifications. One powerful approach is to use *symbolic indexing*, where a vector of Boolean functions is interpreted as the symbolic representation of a bounded integer. This symbolic integer is then used to index into an array of nodes [1, 9]. This notation provides a powerful technique for specifying and verifying the addressing operations of a memory where the symbolic integer represents an address, and the vector of nodes represents the different memory elements.

For example, the effect of a write operation for a random-access memory can be specified by an assertion:

$$\left[(\vec{\texttt{Ad}}\ \textbf{is}\ \vec{A}) \wedge (\texttt{write}\ \textbf{is}\ 1) \wedge (\texttt{data}\ \textbf{is}\ d) \wedge \mathbf{N}\mathit{true} \Longrightarrow \mathbf{N}(\texttt{M}[\vec{A}]\ \textbf{is}\ d)\right]$$

In this assertion, $\vec{\texttt{Ad}}$ is a vector of the p nodes forming the address inputs to the memory, while $\vec{A}$ is a vector of p Boolean variables. $\vec{\texttt{M}}$ is a vector of 2^p nodes forming the memory elements. Informally, the assertion states: "given address and data values A and d on the inputs, a write operation will cause data d to be stored in memory location A. Note that we have interpreted the "next-time" operator as denoting a complete cycling of the memory. In practice we actually operate the memory at a phase-level, and use the phase-level timing model described above.

Memory verification illustrates the efficiencies our method gains by partially-ordered system modeling. To verify the above assertion, the verifier would execute a simulation with all memory locations initialized to X, and with the address and data inputs set to Boolean variables, requiring a total of $p+1$ Boolean variables to verify the behavior of a 2^p-bit memory. To check the consequent, it would compare the resulting state of each memory location i with the function $[(i_{p-1} \mathbin{\overline{\oplus}} A_{p-1}) \cdots (i_0 \mathbin{\overline{\oplus}} a_0)]\,\dot{?}d$, where i_j is the jth bit in the binary representation of i, A_j is the jth element of the vector of variables $\vec{A}$, and $\overline{\oplus}$ represents the Exclusive-Nor operation, i.e., the complement of Exclusive-Or. For example, for a 4-bit memory ($p = 2$), the verification conditions for each memory location would be:

M[0]	M[1]	M[2]	M[3]
$\overline{a}_1\overline{a}_0 \dot{\rightarrow} d \mid \dot{X}$	$\overline{a}_1 a_0 \dot{\rightarrow} d \mid \dot{X}$	$a_1\overline{a}_0 \dot{\rightarrow} d \mid \dot{X}$	$a_1 a_0 \dot{\rightarrow} d \mid \dot{X}$

Full verification of a memory also requires verifying the read operation, as well as verifying that neither operation affects the data in any location other than the one being addressed. All of the operations can be verified by 3 symbolic simulations, none involving more than $2p+1$ variables. We can exploit the large number of "don't care" conditions that arise in the operation of a memory. In verifying memory behavior for a given location, we don't generally care what values were stored in other memory locations. Similar methods can be used to efficiently verify more complex systems containing embedded memories and register arrays, such as microprocessors and data paths.

8.3 User Defined Constructs

With the above extensions, it is more convenient to write specifications. However, any non-trivial specification would still be much too large and obscure to be practical. What is needed is some way of structuring the specification. In the prototype tools we have developed [13, 36] this is

accomplished by using a meta-language [23]. In other words, we use a general purpose language to build up the various constructs that our specification language contain.

In our original prototype system [13] we used a dialect of Lisp as meta-language. When the Lisp program was run, it wrote to a file the verification conditions expressed in a slightly enriched version of the core logic that resulted in the translation of the higher level constructs. This text file was then fed to a modified version of the COSMOS symbolic simulator.

In a more recent system, called Voss[36], developed at the University of British Columbia, the meta language is a dialect of ML[31]. Here, the modified version of the symbolic simulator is incorporated directly in the language and thus the user interacts directly with the evaluator through the ML language. For more details of this system, the reader is referred to [36].

Given that the verification system is embedded in a general purpose language, and the user actually writes code in this language, it is easy to define new extensions. In fact, by writing new functions and procedures it becomes very natural to express the trajectory assertions in a hierarchical way, improving the readability of—and consequently the confidence in—the assertions.

9 Verification Over Other Domains

So far, all our examples have been related to switch-level (and gate-level) verification. On the other hand, the theory was developed using a very general model of systems. The question arises whether there are other domains for which trajectory evaluation is useful. In this section we will discuss one such domain and an application that can beneficially be modeled in the domain.

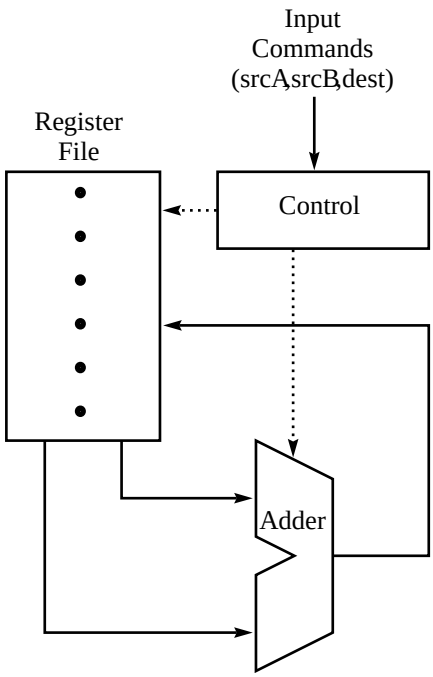

Figure 8: Simple addressable register file with ALU.

Consider verifying the circuit shown in Fig. 8. Intuitively, there are two properties we would like to check:

1. If register A holds some value u and register B holds some value v and we request the circuit to add registers A and B and put the result in register D, then $u + v$ should be stored in register D after the next cycle.

2. If register L stores some value u and we request the circuit to add registers A and B and put the result in register D, where $D \neq L$, then register L should still contain the value u at the end of the next cycle.

The circuit of Fig. 8 can clearly be modeled at a switch-level and be verified using the switch-level model we have used throughout the paper. However, for very wide data path, this could be quite expensive. Also, if the circuit contained a multiplier, rather than an adder, we would very quickly encounter difficulties in carrying out the symbolic evaluation since we would most likely represent the values on the nodes as some kind of ordered binary decision diagram which has difficulties in representing multiplication [10].

What makes the above dependency on the word size unfortunate is that, in some sense, the width of the data path is unrelated to the functionality of the circuit. In particular, the control logic is likely to be independent of the width of the data path. The question arises how to verify the control part for an arbitrary width of the data path. The natural way of verifying the controller by writing a specification in terms of internal control lines is both cumbersome and error prone. What we would like to do is to replace the detailed implementation of the data path with a more abstract, and computationally cheaper, version. If we do so, we split up the verification task into verifying that the abstract version of the data path correspond to the actual data path and that the controller together with the abstract data path works as intended. The first task is quite straightforward since the structure of the abstract data path will likely correspond very closely with the structure of the actual data path. Thus we will focus on the second task. This approach is conceptually similar to the *abstraction* techniques used in temporal logic model checking [17, 38].

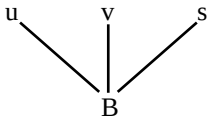

Figure 9: Value domain for data path.

In order to illustrate the idea of using a more abstract domain and corresponding abstract version of the data path, consider the flat domain whose Hasse diagram is shown in Fig. 9. Intuitively, u and v are used to represent arbitrary values and s is used to represent the sum of u and v. The value B is used to denote an unknown value. A possible next-state function for the adder and a possible next-state function (R_i) for one of the of the register words when the write enable signal (W) is 0, 1, and X respectively, are shown in Fig. 10. It is easy to convince oneself that the next-state function is monotone.

The complete lattice for the circuit can now be formed in the same way as for the switch-level model discussed in Section 3, i.e., we form the cross product of all the subcomponents' domains and then add an artificial top element. Also, the next-state function can be derived by extending the individual excitation functions to this extended domain. It is easy to verify that the obtained lattice and next-state function indeed satisfies our requirements for being a model structure. The only remaining missing piece is now some simple predicates for this domain. We will use the obvious ones: n_i **is** u, n_i **is** v, and n_i **is** s, where n_i is a node name in the circuit. Note that "node" in this

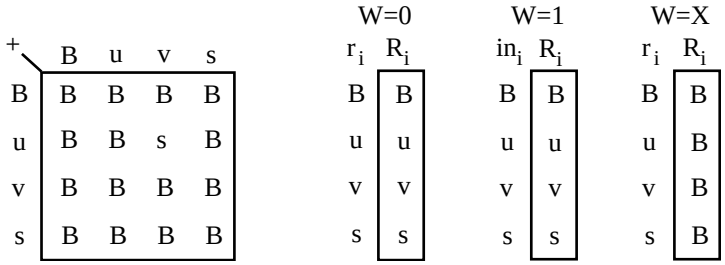

Figure 10: Monotone next-state functions.

context does not correspond to any single electrical node of the circuit but to collections of signals forming data words.

In order to write trajectory assertions that can check the two properties mentioned above, the following shorthands are useful. Let $\vec{I}$, $\vec{J}$, $\vec{K}$, and $\vec{L}$ each denote vectors of p Boolean variables indicating possible address values, where p is the number of bits in an address. Define $Operate(\vec{I}, \vec{J}, \vec{K})$ to denote the formula $(\vec{\mathtt{sA}}\ \mathbf{is}\ \vec{I}) \wedge (\vec{\mathtt{sB}}\ \mathbf{is}\ \vec{J}) \wedge (\vec{\mathtt{D}}\ \mathbf{is}\ \vec{K})$, where node vectors $\vec{\mathtt{sA}}$, $\vec{\mathtt{sB}}$, and $\vec{\mathtt{D}}$ denote the address inputs for the control logic. Similarly, let $Stored(\vec{N}, \alpha)$, for α equal to u, v, and s, denote the formula $(\mathtt{R}[\vec{N}]\ \mathbf{is}\ \alpha)$, where $\mathtt{R}$ denotes the set of "nodes" comprising the register file.

With this notation we can express the two desired properties as follows:

$$\left[Operate(\vec{I}, \vec{J}, \vec{K}) \wedge Stored(\vec{I}, u) \wedge Stored(\vec{J}, v) \Longrightarrow \mathbf{N}Stored(\vec{K}, s)\right]$$

and

$$\left[Operate(\vec{I}, \vec{J}, \vec{K}) \wedge Stored(\vec{L}, u) \Longrightarrow \mathbf{N}((\vec{K} \neq \vec{L}) \rightarrow Stored(\vec{L}, u))\right].$$

Here we have actually assumed a unit-delay for the complete cycle. An obvious generalization would adapt the verification conditions to more realistic timing. Note that the complete verification only requires $3 * \log(n)$ Boolean variables for a register file with n words. Also, the verification is independent of the actual width of the data path.

In many ways, the idea of using a flat domain in carrying out the verification is similar to the idea of "generic" specifications [26]. In generic specifications, which relies on using higher-order logic, the actual computation performed by the ALU and the other components in the data path, are simply provided as functions that are not instantiated during the proof of the control logic. In fact, the high-level correctness proof for the circuit of Fig. 8 would be of the form "for every possible function f of proper type, the circuit will read the contents of registers A and B, apply f to these two values, and write the result into register D. Our approach of using a flat domain and using a conservative next-state function can be viewed as Skolemizing the universal quantification in the generic specification and incorporating the computation in the value domain. Thus, the value s we added to the domain, corresponds to $f(u, v)$.

In general, this use of a flat domain for parts of the circuit works well for circuits in which there is a clear distinction between data path and control. The difficult task of verifying the control logic can thus be carried out independently of the width of the data path. Of course, in using higher-level models such as this, one must generate more abstract system models than does our current switch-level circuit analyzer. We leave this task as future research.

10 Conclusions

In terms of mathematical sophistication, the problem solved by our verification 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 system 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.

Despite the limitations of our program, it is suitable for verifying highly complex systems. Recently Beatty has developed a methodology for specifying and verifying systems that uses symbolic trajectory evaluation to carry out the actual verification [2, 3]. In his approach the user specifies the desired system behavior by assertions of the form $[A \Longrightarrow \mathbf{N}C]$, where each timing step represents the completion of a high level operation, such as the execution of an instruction by a microprocessor. The user gives an "implementation mapping" describing how the state values in the specification (e.g., the values of progrmmer-visible registers) are realized within the circuit, including the exact signal timing. The verifier combines these high level assertions with the implementation mapping to generate symbolic assertions that are then evaluated on the circuit model. His method can handle pipelined implementations in which operations that are viewed as occuring sequentially in the specification actually execute concurrently in the circuit. He has demonstrated the power of his approach by verifying a representative sample of instructions for an actual microprocessor.

One interesting property of our algorithm, in fact, is that its computational complexity is relatively insensitive to the system 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 random access memory 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.

An interesting question that still is unanswered is whether this type of combination of abstraction and symbolic manipulation can be used in more traditional model checking algorithms. For example, is there some suitable domain for which we can approximate the powerset of the real system by a much smaller complete lattice in such a way that the validity of some temporal formula in the approximate lattice implies the validity of the formula in the real system.

Another open question is how to develop a practical verification methodology using the type of abstract domain verification as was discussed in Section 9. In fact, the general question of what kinds of methodologies can be used for this type of formal verification is largely unanswered.

Acknowledgements

The first author would like to acknowledge the very productive research environment provided by the Integrated Systems Design Laboratory at the University of British Columbia.

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, pp. 98–112.

[2] D. L. Beatty, "A Methodology for Formal Hardware Verification, with Application to Microprocessors," Technical Report CMU-CS-;93-190, Carnegie Mellon University, 1993.

[3] D. L. Beatty, and R. E. Bryant, "Formally Verifying a Microprocessor using a Simulation Methodology," *31st Design Automation Conference*, June, 1994, pp. 596–602.

[4] S. Bose, and A. L. Fisher, "Verifying Pipelined Hardware Using Symbolic Logic Simulation," *International Conference on Computer Design*, IEEE, 1989, pp. 217–221.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] R. E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," *IEEE Transactions on Computers*, Vol. 40, No. 2 (February, 1991), pp. 205–213.

[11] 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.

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

[13] 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, pp. 297–402.

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

[15] 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, pp. 46–51.

[16] 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.

[17] E. M. Clarke, O. Grumberg, and D. E. Long, "Model Checking and Abstraction," *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, Jan., 1992.

[18] 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.

[19] O. Coudert, J.-C. Madre, and C. Berthet, "Verifying temporal properties of sequential machines without building their state diagrams," *Computer-Aided Verification '90*, E. M. Clarke, and R. P. Kurshan, *eds.* American Mathematical Society, pp. 75–84.

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

[21] 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, pp. 271–276.

[22] T. Kam, and P. A. Subramanyam, "Comparing Layouts with HDL Models: A Formal Verification Technique," *International Conference on Computer Design*, IEEE, 1992, pp. 588–591.

[23] M. Gordon, R. Milner, and C. Wadsworth, "Edinburgh LCF", *Lecture Notes in Computer Science*, No. 78, Springer Verlag, 1979.

[24] 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.

[25] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE," *IEEE Transactions on Software Engineering*, Vol. 18, No. 9 (September, 1992), pp. 785–793.

[26] J. Joyce, "Generic Structures in the Formal Specification and Verification of Digital Circuits", *The Fusion of Hardware Design and Verification*, G. Milne, *ed.*, North Holland, 1988, pp. 50–74.

[27] J. Joyce and C. Seger, "Linking BDD-Based Symbolic Evaluation to Interactive Theorem-Proving", *30th Design Automation Conference*, 1993, pp. 469–474.

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

[29] R. P. Kurshan, and K. L. McMillan, "Analysis of Digital Circuits Through Symbolic Reduction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 10, No. 11 (November, 1991), pp. 1356–1371.

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

[31] R. Milner, "A Proposal for Standard ML", *Proceedings of ACM Conference on LISP and Functional Programming*, Austin, TX, Aug. 1984, pp. 184–197

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

[33] 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.

[34] 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.

[35] C. Seger and J. Joyce, "A Mathematically Precise Two-Level Formal Hardware Verification Methodology", Technical Report 92-34, Department of Computer Science, University of British Columbia, December 1992.

[36] C. Seger, "Voss—A formal hardware verification system: User's guide," Technical Report 93-45, Department of Computer Science, University of British Columbia, December, 1993.

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

[38] P. Wolper, "Expressing Interesting Properties of Programs in Propositional Temporal Logic," *Proc. 13th Annual ACM Symposium on Principles of Programming Languages*, Jan., 1986, pp. 184–193.