CSC 2108 - Final Project: Symbolic Trajectory Evaluation

Mihaela Gheorghiu

January 26, 2004

Abstract

The purpose of this work is to survey the Symbolic Trajectory Evaluation (STE) method of formal hardware verification, and present a tool implementing STE.

1 Introduction

Logic simulation has been, for a long time, the dominant approach to the verification of correctness for digital circuits. In this approach, sample logic values are provided at the inputs of a circuit and its behavior in response to these inputs is examined and compared to the expected one. This is essentially a testing, rather than a verification, methodology, as it can discover errors but not prove their absence. If such a proof is sought, simulation has to guarantee that the tested scenarios cover all possible behaviors of a circuit. Simulating a circuit exhaustively *is* possible, but computationally inefficient.

As a simple example, consider verifying that a given circuit correctly implements a logical-AND gate with seven inputs. Exhaustive simulation with explicit logical values requires trying all 2^7 different input vectors of 0s and 1s and comparing the output obtained from the circuit under test for conformance to the truth table of the seven-input logical-AND function. In practice, circuits have large numbers of inputs, so this approach is clearly inefficient. Also, exhaustively simulating *sequential*, rather than *combinational* (as in our example), circuits, is even more difficult, if at all possible, since in sequential circuits the output depends not only on current inputs, but also on previous ones, so it is not sufficient to simulate all possible input vectors, it is also important to simulate all possible *sequences* of input vectors.

Using an enriched logic for simulation can help. For instance, using Xs in addition to 0s and 1s, reduces the number of input vectors that have to be tested. In our example, instead of 2^7 input patterns, we need only eight: $0XXXXXX$, $X0XXXXX$, $XX0XXXX$, $XXX0XXX$, $XXXX0XX$, $XXXXX0X$, $XXXXXX0$, for which the expected output is 0, and 1111111, for which the expected output is 1, where X stands for "don't care". For some sequential circuits, it has been noted [1] that setting the internal signals of such a circuit to all Xs before trying a new input vector, avoids the problem of having to try all possible sequences of inputs. The disadvantage of such multi-valued simulation is that it may be more pessimistic than necessary, producing false negatives.

Another way in which exhaustive simulation can be made feasible is by using *symbolic*, rather than explicit, logical values. For our example, this reduces to testing only one input pattern: a, b, c, d, e, f, g, and comparing the output obtained to the Boolean expression $a \wedge b \wedge c \wedge d \wedge e \wedge f \wedge g$. This approach involves symbolic manipulation and comparison. While the benefits of such simulation have been recognized early [2], it has not been made practical until the advent of efficient data structures and algorithms for symbolic manipulation and evaluation of Boolean expression such as BDDs and their algorithms [3]. The disadvantage of symbolic approaches is that they do not scale well.

Symbolic Trajectory Evaluation (STE) is a method for formal hardware verification that has grown from the combination of multi-valued simulation *and* symbolic simulation. It started with the observation that the two approaches could compensate each other: having more than two logical values reduces the number of symbolic variables and hence the size of the BDDs needed in symbolic simulation, and having symbolic manipulations eliminates some of the false negatives from multi-valued simulation. One source of false negatives when using Xs is that X OR $\neg X = X$, since the three-valued logic does not obey the law of excluded middle, but with symbolic Boolean variables a OR $\neg a = 1$ can be computed with BDDs.

STE has been successfully used in industry (Intel, IBM, Motorola) for the verification of large-scale designs. One key to its success is the fact that, being simulation-based, it has much of the "black-box" character of simulation, thus being intuitive, and needing little expert knowledge. This is also one of the characteristics that make it similar to standard model-checking. There are many other and more technical similarities between the two, some of which are explained in this paper.

The rest of the report is structured as follows: Section 2 contains the basic theory underlying STE. Section 3 outlines some of the extensions that have been applied to STE, up to its current version. Section 4 presents and exemplifies the academic version of Forte, the tool from Intel implementing STE. Section 5 concludes our work.

2 Basic STE Theory

This section presents the core STE theory, following [4]. Our presentation is focused more on explaining the results in [4], rather than just reproducing them.

2.1 System model

For STE, a system (circuit) is modelled by a pair $\mathcal{M} = \langle(\mathcal{S}, \leq), Y\rangle$, where $(\mathcal{S}, \leq)$ is a complete lattice of *states*, and $Y : \mathcal{S} \to \mathcal{S}$ is a monotonic, deterministic *next-state function*.

To illustrate this, consider a circuit that simply consists of one inverter, shown in Figure 1.

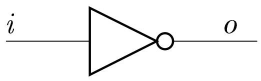


Figure 1: Inverter.

We identify the input and output of the inverter as the *nodes* of interest (state variables) in the system and denote them i, o. We consider the domain of these variables to be $\mathcal{T} = \{0, X, 1\}$, with the "information" partial order pictured in the Hasse diagram in Figure 2. Value X represents a signal whose value is unknown.

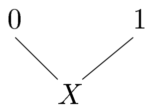


Figure 2: Information order.

This partial order induces a partial order on $\mathcal{T} \times \mathcal{T}$, defined pointwise. The bottom element $\perp$ of $\mathcal{T} \times \mathcal{T}$ is (X, X); in order to complete the lattice, we add a top element $\top$, that can be viewed as an "overconstrained" state, in which both state variables are 0 *and* 1 at the same time. Hence, in this case, $\mathcal{S}$ is $\mathcal{T}^2 \cup \{\top\}$. We denote each element s in $\mathcal{T}^2$ as a pair (s_i, s_o) of values for the input and the output, respectively. Each node has an *excitation* that expresses the constraint on the value that node can take in a next state. Since there is no such constraint for i, its excitation y_i is defined as $y_i(s) = X$, for all $s \in \mathcal{T}^2$. For o, we have $y_o(s) = \neg s_i$, for all $s \in \mathcal{T}^2$ (where $\neg 0 = 1, \neg 1 = 0, \neg X = X$). The next-state function Y in our model is defined as: $Y(s) = (y_i(s), y_o(s))$, if $s \in \mathcal{T}^2$, or $Y(s) = \top$, if $s = \top$.

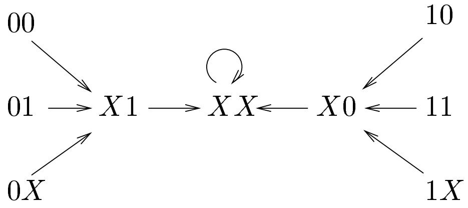


Figure 3: Inverter model.

It can easily be verified that Y is monotonic. The states and next-state function of our circuit are pictured in Figure 3. A state is shown as a pair without parenthesis and commas, and its successor state assigned by the next-state function is pointed to by an arrow; state $\top$ is not shown, as it is unreachable.

In general, the excitations of nodes in a circuit are Boolean functions corresponding to the functionality of the nodes. These excitations are extended to ternary functions (functions on $\mathcal{T}$) so that they agree with their Boolean counterparts on binary values, and are monotonic with respect to the information order. These requirements apply to any multi-valued and partially ordered domain that may be chosen instead of $\mathcal{T}$, for instance when modeling circuits at higher levels than the gate or switch level.

2.2 Trajectories

To describe the behavior of a system (model) $\langle(\mathcal{S}, \leq), Y\rangle$ over time, we consider the set $\mathcal{S}^\omega$ of all infinite sequences of its states. Not all such sequences are possible behaviors - only the ones that obey the next-state function are, and they are called *trajectories*. Formally, a sequence $\sigma = \sigma^0\sigma^1 \ldots \in \mathcal{S}^\omega$ is a trajectory iff $Y(\sigma^i) \leq \sigma^{i+1}$ for $i \geq 0$. For our example, $(1, X),\ (X, 0),\ \bot,\ \bot, \ldots$, for instance, is a trajectory, since $Y(1, X) = (X, 0)$, $Y(X, 0) = \bot$, and $Y(\bot) = \bot$. On the other hand, $(0, 0),\ \bot,\ \bot, \ldots$ is not, since $Y(0, 0) = (X, 1)$, and it is not true that $(X, 1) \leq \bot$. The partial order on $\mathcal{S}$ is extended to $\mathcal{S}^\omega$ pointwise, that is: for $\sigma = \sigma^0\sigma^1 \ldots$, and $\tau = \tau^0\tau^1 \ldots$ in $\mathcal{S}^\omega$, $\sigma \leq \tau$ iff for all $i \geq 0$ we have $\sigma^i \leq \tau^i$.

2.3 Trajectory formulas

For reasoning about behaviors, STE allows one to write *trajectory formulas* which are formulas of a restricted temporal logic.

The simplest trajectory formula is a *simple predicate* $p : \mathcal{S} \to \{\textbf{true}, \textbf{false}\}$ that actually applies to a single state. Let $\mathrm{glb}S$ denote the greatest lower bound of set $S \subseteq \mathcal{S}$ with respect to $\leq$. Predicate p is simple iff it is monotonic and glb-closed, *i.e.*, if we denote $s_p = \text{glb}\{s \in \mathcal{S} \mid p(s) = \textbf{true}\}$, then $p(s_p) = \textbf{true}$. State s_p is thus the "weakest" state for which p holds, and p holds for any state that is above s_p, so s_p characterizes p. For these reasons, we call s_p the *defining state* of p (in [4] it is called defining value). For our inverter, a simple predicate would be i *is* 0. Its defining state is $(0, X)$. A simple predicate is true of a trajectory if it is true of its initial state.

Recursively, the conjunction of two trajectory formulas is also a trajectory formula, with the usual semantics, and $\mathbf{N}\ \varphi$ is a trajectory formula, where φ is a trajectory formula. The informal meaning of $\mathbf{N}\ \varphi$ is "next time φ". $\mathbf{N}\ \varphi$ is true of a trajectory $\sigma^0\sigma^1\ldots$ iff φ is true of $\sigma^1\ldots$. Nothing else is a trajectory formula. The depth of a trajectory formula is the maximum number of nested $\mathbf{N}$ operators occurring in the formula, plus one.

The notion of defining state for a simple predicate extends to sequences of states for trajectory formulas. Given a trajectory formula φ, there exists a unique *defining sequence* $\delta_\varphi \in \mathcal{S}^\omega$ such that for any trajectory σ, φ is true of σ iff $\delta_\varphi \leq \sigma$. Thus a trajectory satisfies φ iff that trajectory is "above" the defining sequence δ_φ of the formula. This is an important result that facilitates testing whether a given trajectory satisfies a given formula. The theory provides rules for the explicit construction of the defining sequence for any trajectory formula; basically, the construction uses the defining states of simple predicates as building blocks, and works recursively on the structure of the formula. In our example, for instance, $\delta_{(i \text{ is } 0)} = (0, X),\ \bot,\ \ldots$.

It is important to note that the defining sequence of a trajectory formula is not necessarily a trajectory itself. But the theory further provides an explicit construction that, given a trajectory formula and its defining sequence, it obtains a *defining trajectory* of the formula; the idea behind the construction is to impose the next-state function on the defining sequence, thus transforming it into a trajectory. For our example, the trajectory obtained from the defining sequence given above is: $\tau_{(i \text{ is } 0)} = (0, X),\ (X, 1),\ \bot,\ \ldots$. Hence, for any trajectory formula, all the trajectories satisfying it are those "above" its defining trajectory.

2.4 Specification and verification

Verification using STE consists of checking whether a given model satisfies trajectory assertions of the form $A \Rightarrow C$, where A, C are trajectory formulas. Such an assertion is true of a model if any trajectory that satisfies A, also satisfies C. In other words, all the trajectories satisfying A are among those satisfying C. The specification of a system is given by a set of trajectory assertions. For our inverter, the specification is: $(i \text{ is } 0) \Rightarrow \mathbf{N}\ (o \text{ is } 1)$ and $(i \text{ is } 1) \Rightarrow \mathbf{N}\ (o \text{ is } 0)$.

The theory of STE, as described above, facilitates the verification of such specifications. For any assertion $A \Rightarrow C$, the defining sequence δ_C for C and the defining trajectory τ_A are constructed. This construction is the "simulation" part of the method, since the defining trajectory is obtained by starting from a certain state and applying the next-state function repeatedly, just like simulating a circuit. Then, checking the assertion reduces to checking $\delta_C \leq \tau_A$, justified as follows: since all the trajectories satisfying A are those "above" τ_A, and all the trajectories satisfying C are those "above" δ_C, all the trajectories satisfying A are among those satisfying C iff $\delta_C \leq \tau_A$. Since both δ_C and τ_A are infinite sequences, this would seem as an infinite test, but a result of the theory guarantees that the test needs to examine only the prefixes of the two sequences that are as long as the depth of C, since it is shown that the rest of δ_C consists of all $\perp$s.

3 Extensions

The theory described in the previous section is the kernel of the STE method. Here, we present the most important extensions to the method that have been applied since its introduction, up to its current version.

3.1 Symbolic formulation

As described above, the STE theory does not allow Boolean variables to occur in states, nor in trajectory formulas. In the symbolic version of the theory such variables can occur, rendering greater efficiency: a symbolic assertion has the expressivity of several non-symbolic assertions and verifying such an assertion by symbolic simulation replaces the need for large amounts of case analysis. For instance, the symbolic specification of our inverter contains only one assertion: $(i \text{ is } x) \Rightarrow \mathbf{N}\ (o \text{ is } \neg x)$, where x is a Boolean variable. Note that only Boolean variables are allowed, and not multi-valued ones.

The symbolic formulation of the STE theory has symbolic states, next-state function, sequences of states, trajectories, trajectory formulas, and assertions. They are symbolic in that they may contain Boolean variables. In general, any non-symbolic quantity defined in the previous section, be it a value, a function, an operation, or a relation, has a symbolic version that is a function mapping the truth assignments to (interpretations of) Boolean variables, to non-symbolic quantities of the same type. Values 0, 1, X, and simple predicates have trivial symbolic counterparts: the symbolic version of such a non-symbolic quantity is the constant function returning the same, non-symbolic quantity, regardless of the truth assignment. For instance, a symbolic X is the function that maps any truth assignment to

X, and a symbolic simple predicate is the function that maps any truth assignment to that predicate (no variables are allowed in simple predicates).

A symbolic state is a *function* mapping between the truth assignments to its Boolean variables and states. For instance, the symbolic state $(x, \neg x)$ of our inverter maps $x = 0$ to state $(0, 1)$, and $x = 1$ to state $(1, 0)$. Equivalently, a symbolic state can be interpreted as a set of non-symbolic states. In our example, $(x, \neg x)$ represents the set $\{(0, 1), (1, 0)\}$. Consequently, a symbolic sequence of states is a sequence of symbolic states, which is a function mapping between truth assignments to its Boolean variables and sequences of states. In other words, a symbolic sequence of states represents a set of non-symbolic sequences of states. The next-state function is extended to symbolic states accordingly, and so are trajectories. For our inverter, the successor (next state) of symbolic state $(x, \neg x)$, is the function that maps $x = 0$ to state $(X, 1)$ (*i.e.*, to the successor of $(0, 1)$, and $x = 1$ to $(X, 0)$ (*i.e.*, to the successor of $(1, 0)$). We could equivalently consider the successor of $(x, \neg x)$ to be the set of states $\{(X, 1), (X, 0)\}$, or the symbolic state $(X, \neg x)$. Note that this symbolic state can be computed directly from $(x, \neg x)$ using the available excitations: that of the input is X, and that of the output is the negation of the input, hence $\neg x$.

A symbolic trajectory formula is itself a function that maps truth assignments to its Boolean variables, to trajectory formulas. The syntax is only slightly changed; a symbolic trajectory formula is, as before, either a simple predicate, or the conjunction of two symbolic trajectory formulas, or of the form $\mathbf{N}\ F$, where F is a symbolic trajectory formula, or, unlike before, of the form $e \rightarrow F$, where e is a Boolean expression and F, a symbolic trajectory formula. The latter formula is true of a trajectory under a given truth assignment iff e logically implies that the trajectory satisfies F, under that truth assignment. This new definition allows a convenient syntactic shorthand, n is e, where n is a node and e is a Boolean expression, to stand for $e \rightarrow (n \text{ is } 1) \wedge \neg e \rightarrow (n \text{ is } 0)$. This syntactic shorthand was used in the symbolic version of the inverter specification given above.

Trajectory assertions are, as before, of the form $A \Rightarrow C$, only that now A and C are symbolic trajectory formulas. The verification algorithm checks whether the assertion is true under all possible truth assignments to the Boolean variables that occur in A and C. The results facilitating verification in the non-symbolic case extend naturally to the symbolic version. Each symbolic trajectory formula has a defining symbolic sequence and a defining symbolic trajectory, that are the symbolic extensions of their counterparts defined in the previous section. Then checking whether the system satisfies a symbolic assertion $A \Rightarrow C$ amounts to verifying $\delta_C \leq \tau_A$ under all possible truth assignments, where δ_C is the defining symbolic sequence for C, τ_A is the defining symbolic trajectory for A.

To illustrate the symbolic verification algorithm, recall the symbolic specification (i is x) $\Rightarrow$ $\mathbf{N}$ (o is $\neg x$) of our inverter. The defining symbolic trajectory for the antecedent is $(x, X),\ (X, \neg x),\ (X, X), \ldots$, and the defining symbolic sequence for the consequent is $(X, X),\ (X, \neg x),\ (X, X), \ldots$. Since $X \leq x$ regardless of whether x is 0 or 1, we conclude that the inverter satisfies its specification.

The symbolic theory can be implemented using BDDs. In order to represent the ternary domain, each symbolic quantity is represented by a *pair* of BDDs (this is known as *dual-rail* encoding):

- 0 is $(0, 1)$
- 1 is $(1, 0)$
- X is $(0, 0)$
- any Boolean variable a is $(a, \neg a)$
- for any values or Boolean variables $a = (a_1, a_2), b = (b_1, b_2)$, any operation on a, b is a pair of Boolean operations: $a \wedge b$ is $(a_1 \wedge b_1, a_2 \vee b_2)$, $a \vee b$ is the dual of $a \wedge b$, $\neg a = (a_2, a_1)$, glb $\{a, b\}$ is $(a_1 \wedge b_1, a_2 \wedge b_2)$, and lub is the dual of glb.

Note that checking whether two symbolic states a, b are related as $a \leq b$ can be implemented as testing glb $\{a, b\}$ and a for equivalence, which means testing two pairs of BDDs for equivalence. Thus, all operations needed in the STE can be implemented using known BDD operations. In actual implementations of STE, if the symbolic defining sequence of the consequent is not less than or equal to the symbolic defining trajectory of the antecedent under all truth assignments, as required, the verification algorithm produces the BDD representing those truth assignments for which the required relationship holds (see Section 4 and the Appendix).

3.2 Extending the logic

The specification language described so far is very restricted in terms of syntax and semantics of trajectory formulas and assertions. In this section we outline several extensions to this language, as introduced in [4, 5].

In [4], beside checking simple assertions of the form $A \Rightarrow C$, the verification algorithm can handle also assertions of the forms (defined recursively) (1) $(A \Rightarrow C);\ G$, or (2) $(A \Rightarrow C)^*;\ G$, where G is a trajectory assertion. An assertion of type (1) holds if $A \Rightarrow C$ holds, and G is satisfied by the suffix of any trajectory that satisfies A, obtained by removing the prefix of length d of that trajectory, where d is the depth of A (*i.e.*, after removing the prefix matching A, the rest of any trajectory satisfying A satisfies G). The depth of C is ignored, since it can always be made equal to that of A by inserting conjunct ($\mathbf{N}$ **true**) into A or C, nested inside as many $\mathbf{N}$s as necessary. To check such an assertion, the algorithm proceeds in the usual way for $A \Rightarrow C$. The verification of G proceeds recursively, and is facilitated by a result which guarantees that there exists a defining trajectory for any trajectory formula with respect to *any initial state* of the system. For simple assertions, the initial state was taken to be $\perp$. In checking G here, the initial state is taken to be the $d+1$-st state in the defining trajectory of A. The initial state matters only for defining trajectories, and not for defining sequences.

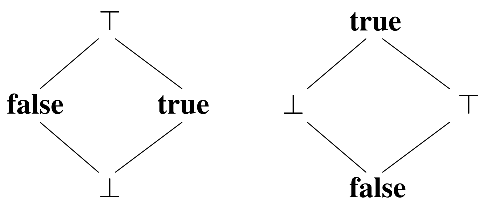
An assertion of type (2) holds iff G holds and for any $i \geq 1$, $(A \Rightarrow C)^i;\ G$ holds, where H^i stands for a string of i copies of assertion H separated by ";". To check such an assertion recursively, with respect to an initial state z, G is checked with initial state t, and $(A \Rightarrow C)$ is checked with respect to the same t, with $t =$ Gfp$\boldsymbol{\xi}$.glb $\{z, u(\boldsymbol{\xi})\}$, where Gfp denotes greatest fixed-point, and $u(\boldsymbol{\xi})$ is the $d+1$-st state in the defining trajectory for A with respect to initial state $\boldsymbol{\xi}$. The recursion starts with $z = \perp$. The fixed-point is the weakest state reachable from z after any number of matchings to A and it represents all the states the system could be in after satisfying A any number of times. Since $\mathcal{S}$ is a finite lattice and glb $\{z, u(\boldsymbol{\xi})\}$ is a monotone function, the fixed-point computation is done iteratively, starting from $\boldsymbol{\xi}_0 = \top$.

Because of this fixed-point computation, the algorithm may give overly pessimistic answers, in the case of assertions containing *. For assertions not containing *, it is guaranteed, however, that the algorithm gives an exact answer, that cannot be a false positive or a false negative. A symbolic version of the algorithm handling these extended forms of assertions is also defined, similarly as before.

Significant extensions to the theory are made in [5]. The temporal logic for trajectory formulas is extended to include *negation*, and the strong *Until* operator, with the usual semantics. These extended trajectory formulas take values from a *four-valued* truth lattice $\mathcal{Q}$ that, beside **true** and **false** contains $\perp$ (*unknown*), and $\top$ (*inconsistent*). Value $\perp$ is introduced in order to distinguish between a formula being definitely true or false, and not having enough information to decide the truth or falsity of a formula. Value $\top$ is necessary to represent contradictory information that would allow both the truth and falsity of a formula to be inferred. This multi-valued logic corresponds more naturally than a two-valued one to models where system variables are themselves multi-valued. A slight change is also made in the modeling of systems. If before system variables where three valued $0, X, 1$, now they are four-valued $0, X, 1, Z$, where Z stands for inconsistent. Before, only one state, $\top$, was inconsistent, now individual variables can be inconsistent. The mathematical structure of $0, X, 1, Z$ is the same as that of $\mathcal{Q}$, with the information order, but the semantics is different; the former represents state values, whereas the latter, truth values.

Two orders are considered on $\mathcal{Q}$, the information order, and the truth order, pictured in the Hasse diagrams of Figures 4(a) and 4(b), respectively.

Conjunction, disjunction and negation in $\mathcal{Q}$ are computed by taking the meet, join, and weak negation ($\neg\top = \top, \neg\perp = \perp, \neg\,$**true** $=$ **false**$, \neg\,$**false** $=$ **true**) respectively, under the truth order. Thus defined, they agree with their Boolean



(a) (b)

Figure 4: Four-valued lattice $\mathcal{Q}$.

counterparts on **true** and **false** arguments, and are monotonic with respect to the information order.

In this new context, a simple predicate has a defining state for each of the four possible truth values, so it has a *defining set of states*. We no longer have a unique defining sequence, and a unique defining trajectory, but a *set of defining sequences* and a *set of defining trajectories* for each trajectory formula. The assertions handled by the new verification algorithm are of the form $g \Rightarrow h$, where g, h are trajectory formulas over the extended temporal logic. The truth of such an assertion can be established in two ways, depending on the handling of inconsistency: by checking that for any trajectory that satisfies g with value (exactly) **true**, that trajectory also satisfies h with value (exactly) **true**, or by checking whether any trajectory that satisfies g with value *at least* **true** also satisfies h with value *at least* **true**. The latter version turns out to be easier to check, and amounts to verifying that for any trajectory τ in the defining set of trajectories for g there exists a sequence σ in the defining set of sequences for h such that $\tau \leq \sigma$. From the truth of the latter version of an assertion, the truth of the former version can be determined, by keeping track of inconsistency. A symbolic version of this extended theory is also defined, similarly as in the case of the basic STE theory. The representation of symbolic quantities by pairs of BDDs is as before, with the addition that the $\top$ value in $\mathcal{Q}$ is represented by $(1, 1)$ ($\perp$ is as X before).

3.3 Generalized STE

Here we present the extensions included in the current version of the STE [6, 7].

Ever since its introduction, the STE method has paralleled the standard model-checking algorithms for hardware verification. It has been recognized that STE is, itself, a model-checking technique since it checks properties of models of systems. The multi-valued models it works with are just abstractions of concrete models [8]. STE is, however, a special-purpose, restricted technique, since it can only

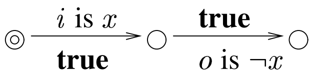


Figure 5: Inverter assertion graph

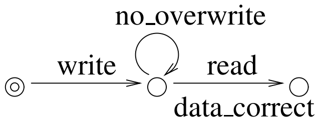


Figure 6: Sample GSTE assertion graph

check properties with a very restricted syntax. The latest work on the STE has been trying to relax the method and make it into a more general model-checking technique, while preserving its basic approach and efficiency. This has led to the current version of the method, called Generalized STE (GSTE).

GSTE generalizes the assertions that the verification algorithm can check to *assertion graphs*. A classical STE assertion of the form $A \Rightarrow C$ can be viewed as a labeled linear directed graph. For instance, our inverter specification $(i \text{ is } x) \Rightarrow \mathbf{N}\ (o \text{ is } \neg x)$ from before is pictured as a graph in Figure 5.

Each edge in the assertion graph represents a time unit and is labeled with both an antecedent label (above the edge), and a consequent label (below the edge), namely those state predicates from the antecedent and consequent, respectively, corresponding to that time unit. In GSTE, any such assertion graph, not only linear ones, is considered; the only requirement is that the graph have a finite set of vertices and an initial vertex. An example of a simple GSTE assertion graph (with high-level predicates and **true** labels not shown) is in Figure 6. It expresses the informal property that "if some data is written now, then the next read operation will get the correct data, as long as there is no overwriting in between".

Each finite path in a GSTE assertion graph is an STE assertion. Three different satisfaction definitions and corresponding verification algorithms have been introduced. The *strong* satisfaction rule stipulates that a model satisfies an assertion graph iff it satisfies every finite path (STE assertion) in the graph. The *normal* satisfaction rule states that a model satisfies an assertion graph with respect to a set of terminal graph vertices if for every finite path (STE assertion) in the graph that ends at a terminal vertex, and every finite trace in the model of the same length as that path, if the trace satisfies the antecedent sequence on the path, then it also satisfies the consequent sequence. The difference between the former satisfaction definition and the latter is that in the former each consequent label can depend only on past and present antecedent labels, whereas in the latter it can depend on future antecedents, or equivalently, an antecedent can imply a consequent in the past. Normal satisfaction thus overcomes the limitation of simulation as a ”forward” approach.

Finally, by introducing the notion of ”fair” edges, *fair* satisfaction is defined as: a model fairly satisfies an assertion graph with respect to a set of fair edge sets if for every infinite path visiting each fair edge set infinitely often, and every infinite trace of the model, if that trace satisfies the antecedent sequence on the path, then the trace also satisfies the consequent sequence. With the notion of fairness, it is shown that the GSTE algorithm can check all ω-regular properties; [7] points to [9] for a proof of a theorem stating that every Streett automaton has an equivalent assertion graph. The verification of normal and that of fair satisfaction, both eventually reduce to that of strong satisfaction, and that of strong satisfaction combines the standard STE algorithm with fixpoint computations.

4 Forte

Forte is the formal verification environment used at Intel, that implements the STE theory. Very little information about Forte was available until recently (January 2003) when an academic version of it was released. This is a very restricted version of the real Forte, with many missing libraries. In this section, we demonstrate the use of this version of Forte for the specification and verification of a two-bit counter. Whenever we refer to Forte in the following, we mean the academic version.

4.1 Models in Forte

An input to Forte is a description of a circuit in Exlif (Extended Logic Interchange Format) - a format based on the Berkeley Logic Interchange Format (BLIF). This description contains the low-level structure and behavior of the circuit, as a truth-table or Boolean expression. Our counter is a simple sequential circuit that goes through the sequence of states/values $00, 01, 10, 11, 00, \ldots$, has no designated initial state, and changes state on each rising edge of the clock. For this counter, the Exlif file is listed below.

```
.model counter
.inputs in[1:0]
.outputs out[1:0]
.clock c
```

```
.latch in[1:0] out[1:0] re c
.names out[1] out[0] in[1]
01 1
10 1
.names out[1] out[0] in[0]
00 1
10 1
.end
```

The header is self-explanatory - it gives a name for the model, and declares its interface, consisting of inputs, outputs, and a clock. In our case there are two bits of input which dictate the next state (value) of the counter, and two bits of output, that constitute its current state (value). Both input and output are declared as two-bit vectors in the file. The circuit is essentially a two-bit latch, declared as such - the latch declaration specifies the input and output of the latch to be each two-bit wide, the condition for the state change, "re", to be the rising edge of the clock, and the clock signal to be the one introduced in the header. The semantics of this declaration is that whenever the clock changes from low or 0 to high or 1, the output of the latch takes the value of its input. Listed next, with the `.names` clauses, is the truth table for the latch, more specifically, those conditions that cause each next-state bit to be set to 1. It is understood that the rest of the conditions cause the state bit to be reset to 0. This is just a succinct form of the more usual truth-table for a two-bit counter:

current_state out[1]out[0]	next_state in[1]in[0]
00	01
01	10
10	11
11	00

The Exlif format also allows modular descriptions - a file may contain many models, and one could use several instances of one model inside another. To illustrate this with our counter, here is another, equivalent description:

```
.model counter
.inputs in[1:0]
.outputs out[1:0]
.clock c
.subckt onebit b1 clk=c i=in[1] o=out[1]
.subckt onebit b0 clk=c i=in[0] o=out[0]
.names out[1] out[0] in[1]
```

```
01 1
10 1
.names out[1] out[0] in[0]
00 1
10 1
.end

.model onebit
.inputs i
.outputs o
.clock clk
.latch i o re clk
.end
```

Before inputting it to Forte, an Exlif model is translated into Forte's internal format by the `nexlif2exe` command. For our counter, we run

```
nexlif2exe 2bcounter.exlif
```

where `2bcounter.exlif` is the Exlif file containing the counter model. As a result of this command, file `2bcounter.exe` is produced. The user interacts with Forte through FL, a high-level interpreted functional language. With FL commands, a model gets loaded into FL and represented internally as a Finite State Machine (FSM). That FL sequence for our counter is:

```
let model = load_exe "2bcounter.exe";
model;
```

There are various commands available in FL for model exploration - to inquire about the number of nodes (variables of interest) in the model, their types, connections, functionality, to retrieve a list of nodes, etc. We illustrate some of these commands on our model in the Appendix. There are also several ways to manipulate models - to extract a sub-model from a given model, rename the interface signals of a model, merge two models, etc.

4.2 STE in Forte

This academic version of Forte has very limited verification capabilities, with a very restricted use of the STE. The command that calls STE has the following syntax:

```
STE [<options>] <model> <weak> <antecedent> <consequent>
     <trace>
```

where the most important arguments are

- `<model>` which is the model to be simulated

- `<antecedent>`, that describes the initial conditions of the simulation, and is a list of quintuples of the form `(<guard>, <node>, <value>, <from>, <to>)` with the semantics: if the condition `guard` holds, the given `<node>` is assigned the value `<value>` from time `<from>` to time `<to>`

- `<consequent>`, which describes the expected result of the simulation, and is a similar list of quintuples as the antecedent, with the different semantics that, when the guard is true, then after running STE the given node is expected to have the given value from time `<from>` to time `<to>`

- `<trace>`, that is a list of tuples specifying the nodes whose values we want to trace during the simulation, and the time frames within which we wish to trace them

The times specified with this syntax are not in terms of a clock of the model, but in terms of the internal clock of STE. This syntax corresponds to assertions of the form $A \Rightarrow C$, where A is the antecedent here, and C is the consequent. Explicit reference to time steps takes the place of next operators. This syntax comes naturally in a simulation context (see illustration below). If the corresponding assertion is definitely true, then STE returns `T`; if it is definitely false, STE returns `F`; otherwise, it returns the truth assignments that would make the assertion true, in the form of a Boolean expression.

We have verified several antecedent-consequent pairs for our counter. We decided to trace all the nodes in our model, between time 0 and time 2, so we defined:

```
let trace = [("out[1]", 0, 2), ("out[0]", 0, 2),
             ("in[1]", 0, 2), ("in[0]", 0, 2),
             ("c", 0, 2)];
```

First, we check whether the counter wraps around correctly from 3 to 0:

```
let ant = [(T, "out[1]", T, 0, 1),
           (T, "out[0]", T, 0, 1),
           (T, "c", F, 0, 1),
           (T, "c", T, 1, 2)];
let cons = [(T, "out[1]", F, 1, 2),
            (T, "out[0]", F, 1, 2)];
STE "" model [] ant cons trace;
```

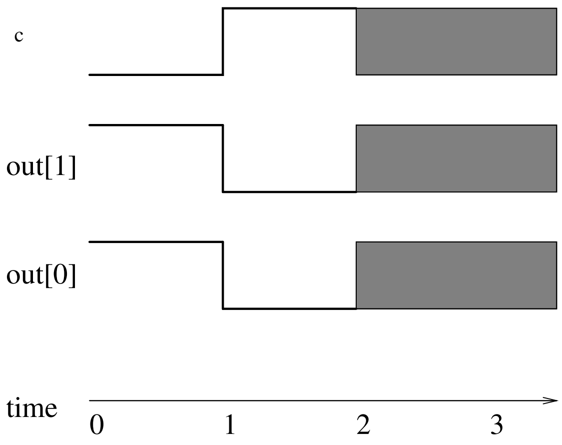


Figure 7: Waveform representation of STE command.

Note that we have to make the rising clock edge explicit.

A simulation-style waveform picture for this example, similar to those shown by conventional simulators, and motivating the syntax of the STE command is in Figure 7. The grey areas in the figure represent “don’t care”s.

Next, we check whether the least significant bit of the counter correctly toggles at every step. Note that this a symbolic simulation, using a variable `a`, unlike the previous one. The syntax `variable <string>` is used to create a BDD corresponding to a variable whose name is given by the string. Forte has several commands for explicit BDD query, manipulation and control, not illustrated here. Here we show only logical operations on BDDs.

```
let a = variable ``a'';
let ant = [(T, "out[0]", a, 0, 1),
           (T, "c", F, 0, 1),
           (T, "c", T, 1, 2)];
let cons = [(T, "out[0]", NOT a, 1, 2)];
STE "" model [] ant cons trace;
```

Next we check whether the most significant bit of the counter incorrectly toggles at every step:

```
let a = variable ``a'';
let ant = [(T, "out[1]", a, 0, 1),
           (T, "c", F, 0, 1),
```

```
            (T, "c", T, 1, 2)];
let cons = [(T, "out[1]", NOT a, 1, 2)];
STE "" model [] ant cons trace;
```

With this restricted syntax allowed for STE, we have no direct way of checking whether the counter counts correctly, since we cannot directly translate a high-level property like “if the current value of the counter is x, its next value is (x+1) mod 4” in this syntax. Instead, we check whether the counter increments correctly by verifying it obeys the Boolean expressions we manually derive from its truth table:

```
let a = variable "a";
let b = variable "b";
let ant = [(T, "out[1]", a, 0, 1),
           (T, "out[0]", b, 0, 1),
           (T, "c", F, 0, 1),
           (T, "c", T, 1, 2)];
let cons = [(T, "out[1]", a XOR b, 1, 2),
            (T, "out[0]", NOT b, 1, 2)];
STE "" model [] ant cons trace;
```

We also check whether the counter obeys some other Boolean expressions:

```
let a = variable "a";
let b = variable "b";
let ant = [(T, "out[1]", a, 0, 1),
           (T, "out[0]", b, 0, 1),
           (T, "c", F, 0, 1),
           (T, "c", T, 1, 2)];
let cons = [(T, "out[1]", b, 1, 2),
            (T, "out[0]", a, 1, 2)];
STE "" model [] ant cons trace;
```

The results of all our checks are in the Appendix.

5 Conclusions

Symbolic Trajectory Evaluation is a special-purpose model-checking technique introduced a decade ago, and developed and applied mostly in industry. Since its evolution has been independent of the mainstream model-checking approaches, it is quite difficult to assess its role and place in the spectrum of model-checking techniques. Its relationship with general-purpose model-checking remains an open problem, which even the authors of the STE are actively working on. Only informal claims on this relationship appear in the literature. We are not aware of any systematic, formal study in this direction.

Our survey has attempted to capture the main ideas behind STE and the most important checkpoints in its evolution. The restricted version of Forte has prevented us from undertaking a larger and more interesting modeling and verification task. A fair comparison between STE and other model-checking algorithms on challenging benchmarks will only be possible when a better version of a tool implementing STE becomes available. We hope that our work can serve as a starting point for a future, more in-depth look into STE and its relationship to standard model-checking.

References

[1] R. Bryant. A methodology for hardware verification based on logic simulation. *Journal of ACM*, 38(2):299-328, April 1991.

[2] W. E. Cory. Symbolic simulation for functional verification with ADLIB and SDL. In *Proc. of the 18th ACM/IEEE Design Automation Conference*:82-89, 1981.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677-691, August 1986.

[4] C.-J. H. Seger and R. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147-190, March 1995.

[5] S. Hazelhurst and C.-J. H. Seger. Symbolic trajectory evaluation.*LNCS*, 1287:3-78, 1997.

[6] J. Yang and C.-J. H. Seger. Introduction to generalized symbolic trajectory evaluation. In *Proc. of ICCD 2001*:360-367, September 2001.

[7] J. Yang, C.-J, H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In *Proc. of FMCAD 2002*: 70-87, November 2002.

[8] C.-T. Chou. The mathematical foundation of symbolic trajectory evaluation. In *Proc. of CAV'99*:196-207, July 1999.

[9] J. Yang, C.-J. H, Seger. Generalized symbolic trajectory evaluation. *Intel SCL technical report*, 2000 (submitted for journal publication).

Appendix

Querying the counter model:
- to get a list of all nodes

```
: nodes model;
it::string list
["c","c_##_","in[0]","in[0]_##_","in[1]",
"in[1]_##_","out[0]","out[0]_##_", "out[1]",
"out[1]_##_"]
```

- to inquire about the type of a node

```
: is_latch model "out[1]";

it::bool
T
```

- to get information about the interface of a node

```
: get_latch_interface model "out[1]";
it::string#string#string#string#string
("in[1]","c","","","")
```

- to get information about the functionality (truth table) of a node

```
: get_node_truth_table model "in[1]";

it::string list
["01","10"]
```

Checking that the counter wraps around correctly from 3 to 0:

```
: load "2bccode0.fl";
it::void
Time: 0

Trace started for node: out[1]
    Current value:X

Trace started for node: out[0]
    Current value:X
```

```
Trace started for node: in[1]
    Current value:X

Trace started for node: in[0]
    Current value:X

Trace started for node: c
    Current value:X
.Trace: Node c at time 0: 0
Trace: Node out[1] at time 0: 1
Trace: Node out[0] at time 0: 1
Trace: Node in[1] at time 0: 0
Trace: Node in[0] at time 0: 0
Time: 1
.Trace: Node c at time 1: 1
Trace: Node out[0] at time 1: 0
Trace: Node out[1] at time 1: 0
Trace: Node in[0] at time 1: 1
Time: 2

Trace ended for node: c

Trace ended for node: in[0]

Trace ended for node: in[1]

Trace ended for node: out[0]

Trace ended for node: out[1]
T
```

Checking that the least significant bit toggles:

```
: load "2bccode1.fl";

it::void
Time: 0

Trace started for node: out[1]
```

```
    Current value:X

Trace started for node: out[0]
    Current value:X

Trace started for node: in[1]
    Current value:X

Trace started for node: in[0]
    Current value:X

Trace started for node: c
    Current value:X
.Trace: Node c at time 0: 0
Trace: Node out[0] at time 0: a
Trace: Node in[0] at time 0: !a
Time: 1
.Trace: Node c at time 1: 1
Trace: Node out[0] at time 1: !a
Trace: Node in[0] at time 1: a
Time: 2

Trace ended for node: c

Trace ended for node: in[0]

Trace ended for node: in[1]

Trace ended for node: out[0]

Trace ended for node: out[1]
T
```

Checking whether most significant bit toggles:

```
: load "2bccode2.fl";
it::void
Time: 0

Trace started for node: out[1]
```

```
    Current value:X

Trace started for node: out[0]
    Current value:X

Trace started for node: in[1]
    Current value:X

Trace started for node: in[0]
    Current value:X

Trace started for node: c
    Current value:X
.Trace: Node c at time 0: 0
Trace: Node out[1] at time 0: a
Time: 1
.Trace: Node c at time 1: 1
Trace: Node out[1] at time 1: X

Warning: Consequent failure at time 1 on node out[1]
Current value:X
Expected value:!a
Weak disagreement when:T
Time: 2

Trace ended for node: c

Trace ended for node: in[0]

Trace ended for node: in[1]

Trace ended for node: out[0]

Trace ended for node: out[1]
F
```

Checking whether counter obeys Boolean expressions derived from its truth table:

```
: load "2bccode4.fl";
```

```
it::void
Time: 0

Trace started for node: out[1]
    Current value:X

Trace started for node: out[0]
    Current value:X

Trace started for node: in[1]
    Current value:X

Trace started for node: in[0]
    Current value:X

Trace started for node: c
    Current value:X
.Trace: Node c at time 0: 0
Trace: Node out[1] at time 0: a
Trace: Node out[0] at time 0: b
Trace: Node in[1] at time 0: !b&a + b&!a
Trace: Node in[0] at time 0: !b
Time: 1
.Trace: Node c at time 1: 1
Trace: Node out[0] at time 1: !b
Trace: Node out[1] at time 1: !b&a + b&!a
Trace: Node in[0] at time 1: b
Trace: Node in[1] at time 1: !a
Time: 2

Trace ended for node: c

Trace ended for node: in[0]

Trace ended for node: in[1]

Trace ended for node: out[0]

Trace ended for node: out[1]
T
```

Checking whether counter obeys other Boolean expressions:

```
: load "2bccode3.fl";
it::void
Time: 0

Trace started for node: out[1]
    Current value:X

Trace started for node: out[0]
    Current value:X

Trace started for node: in[1]
    Current value:X

Trace started for node: in[0]
    Current value:X

Trace started for node: c
    Current value:X
.Trace: Node c at time 0: 0
Trace: Node out[1] at time 0: a
Trace: Node out[0] at time 0: b
Trace: Node in[1] at time 0: !a&b + a&!b
Trace: Node in[0] at time 0: !b
Time: 1
.Trace: Node c at time 1: 1
Trace: Node out[0] at time 1: !b
Trace: Node out[1] at time 1: !a&b + a&!b
Trace: Node in[0] at time 1: b
Trace: Node in[1] at time 1: !a

Warning: Consequent failure at time 1 on node out[1]
Current value:!a&b + a&!b
Expected value:b
Strong disagreement when:a

Warning: Consequent failure at time 1 on node out[0]
Current value:!b
Expected value:a
```

```
Strong disagreement when:a&b + !a&!b
Time: 2

Trace ended for node: c

Trace ended for node: in[0]

Trace ended for node: in[1]

Trace ended for node: out[0]

Trace ended for node: out[1]
!a&b
```