Symbolic Trajectory Evaluation in a Nutshell

Tom Melham and Ashish Darbari

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD, England

March 2004

1 Mathematical Preliminaries

We write $\triangleq$ to mean *equals by definition.* We assume familiarity with elementary propositional logic and predicate calculus notation and use the symbol $\supset$ for logical implication. We use lower-case letters (e.g. a, v, x, y_1) for Boolean variables, and upper-case letters (e.g. P, Q) to stand for formulas of propositional logic (i.e. 'Boolean functions'). We write xs to mean a vector of unique variables $x_1, \ldots, x_n$ for indeterminate n and similarly Ps to stand for a vector of formulas $P_1, \ldots, P_n$.

The notation $P[Qs/xs]$ stands for the result of simultaneously substituting the formulas Qs for all occurrences of the Boolean variables xs in P. The notation $P[xs]$ should be taken to mean a formula that contains free occurrences of the distinct Boolean variables xs. In a context in which a formula has been written $P[xs]$, subsequent use of the notation $P[Qs]$ can then be understood to mean the result of substituting the formulas Qs for the variables xs in $P[xs]$.

We also assume familiarity with the notation of naive set theory [1]. If A and B are sets, we write $A{\rightarrow}B$ for the set of all total functions from A to B. We assume that $\rightarrow$ associates to the right, so $A{\rightarrow}(B{\rightarrow}C)$ may be written $A{\rightarrow}B{\rightarrow}C$. Function application associates to the left, so if $f \in A{\rightarrow}B{\rightarrow}C$, $a \in A$, and $b \in B$, then we can write $f\ a\ b$ for $(f\ a)\ b$.

The semantics of symbolic trajectory evaluation uses some elementary concepts of lattice theory [2]. A *poset* $(S, \sqsubseteq)$ is a partial order $\sqsubseteq$ on a set S. If $(S, \sqsubseteq)$ is a poset and $A \subseteq S$, then $x \in S$ is an *upper bound* for A exactly when $a \sqsubseteq x$ for all $a \in A$. A *lower bound* is defined dually. An upper bound x of A is the *least upper bound* of A, written $\sqcup A$, if $x \sqsubseteq y$ for every upper bound y of A. The *greatest lower bound*, written $\sqcap A$, is defined dually. We also write $a \sqcup b$ (read 'a join b') for $\sqcup\{a, b\}$ when it exists and $a \sqcap b$ (read 'a meet b') for $\sqcap\{a, b\}$ when it exists.

A poset $(S, \sqsubseteq)$ is a *complete lattice* iff $\sqcup A$ and $\sqcap A$ exist for all $A \subseteq S$. If S is finite and $a \sqcup b$ and $a \sqcap b$ exist for all $a, b \in S$, then $(S, \sqsubseteq)$ is a complete lattice.

2 STE Model Checking

Symbolic trajectory evaluation [3] is an efficient model checking algorithm especially suited to verifying properties of large datapath designs. The most basic form of STE works on a very simple linear-time temporal logic, limited to implications between formulas built from only conjunction and the next-time operator.[1] In addition, STE is based on *ternary simulation* [6], in which the binary data domain $\{0, 1\}$ is extended with a third value 'X' that stands for 'either 0 or 1, but we don't know which'. As will be seen later, this gives STE a uniform and flexible state-space abstraction mechanism.

These characteristics allow STE to perform property checking more efficiently than conventional model checking algorithms, which operate over more expressive logics like CTL [7]. Of course, there is a trade-off; because STE's logic is simple, it may take more properties to verify the same functionality than in CTL. On the other hand, although the logic of STE seems rather weak, its expressive power is greatly extended by implementing a *symbolic* ternary simulation algorithm [8].

Symbolic ternary simulation uses Boolean variables and propositional formulas to represent whole classes of data values on circuit nodes. The ternary value associated with each node is given by a symbolic data structure whose variables act as parameters. With this representation, STE can combine many (ternary) simulation runs—one for each assignment of values to the variables—into a single *symbolic* simulation run covering them all. The usual implementation uses BDDs [9], but other representations are possible.

The formulas representing values at different circuit nodes can have variables in common, so they can also record interdependencies among node values. Symbolic values therefore greatly increase the expressive power of the limited temporal logic of STE. For example, input-output relations can be extracted from a circuit by using symbolic simulation to derive formulas for the values on output nodes as functions of variables standing for arbitrary values on input nodes. These can then be checked against a specification in the form of some reference formulas provided by the verification engineer.

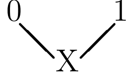
This rest of this section explains the theory of STE model checking in a bit more detail. A full account of the theory can be found in [3] and an alternative perspective is given in [10].

2.1 Circuit Models

Symbolic trajectory evaluation employs a three-valued state model, with values drawn from the set $\mathcal{D} = \{0, 1, \mathrm{X}\}$. The usual binary values 0 and 1 are augmented

[1]Extensions of STE to more expressive logics exist [3, 4, 5], but this paper will focus on the simplest form, which is also the most widely tested on industrial applications.

with an additional value X. This stands for an unknown, which we represent mathematically by a partial order $\leq$ on $\mathcal{D}$ with $X \leq 0$ and $X \leq 1$:



This orders values by information content; X stands for an unknown value and so is ordered below 0 and 1.

2.1.1 States and Sequences

We suppose there is a finite set of circuit nodes $\mathcal{N}$, naming observable points in circuits. You can think of a node as the name of a wire—i.e. just an identifier, such as 'reset' or 'input32'.

To describe the behaviour of a circuit formally, we need to say which values will be present on all its nodes as the circuit evolves over time. A *state* is an instantaneous snapshot of circuit behaviour given by an assignment of values in $\mathcal{D}$ to node names. Formally, a state is represented by a function

$$s \in \mathcal{N} \rightarrow \mathcal{D}$$

that maps each node name to a value.

If the set of circuit nodes is small enough, we can write down specific states just by giving the function explicitly. For example, for $\mathcal{N} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, we might write

$$\{\mathsf{a} \mapsto 0,\ \mathsf{b} \mapsto \mathrm{X}, \mathsf{c}\ \mapsto 1\}$$

for the state in which a is 0, b is X, and c is 1. We can use a more compact notation if we give an ordering on nodes. If the nodes in the example just given are ordered $\mathsf{a} < \mathsf{b} < \mathsf{c}$, we can just write '0X1' to denote the state shown. In what follows, we will feel free to use this notation when convenient.

The ordering $\leq$ on $\mathcal{D}$ is extended point-wise to get an ordering $\sqsubseteq$ on states. We wish this to form a complete lattice, and so we will use a special symbol, $\top$, for the top element of this state lattice. We then define the set of states $\mathcal{S}$ to be $(\mathcal{N} \rightarrow \mathcal{D}) \cup \{\top\}$. The required ordering is defined for states $s_1, s_2 \in \mathcal{S}$ as follows:

$$s_1 \sqsubseteq s_2 \triangleq \begin{cases} s_2 = \top, \text{or} \\ s_1, s_2 \in \mathcal{N} \rightarrow \mathcal{D} \text{ and } s_1(n) \leq s_2(n) \text{ for all } n \in \mathcal{N} \end{cases}$$

The intuition is that if $s_1 \sqsubseteq s_2$, then s_1 may have 'less information' about node values than s_2, i.e. it may have Xs in place of some 0s and 1s.

2.1.2 Abstraction

A state such as 0X1 is really an abstraction. Actual circuit states are always Boolean-valued; each circuit node is either 0 or 1 and there isn't a *value* 'X'. Roughly speaking, you can think of a lattice state as standing for a *set* of ordinary, Boolean-valued states. For example, suppose we have only two circuit nodes, n_0 and n_1. The lattice state 0X can be thought of as an abstraction of the following set of Boolean-valued states

$$\{\{\mathsf{n}_0 \mapsto \mathrm{F},\ \mathsf{n}_1 \mapsto \mathrm{F}\},\ \{\mathsf{n}_0 \mapsto \mathrm{F},\ \mathsf{n}_1 \mapsto \mathrm{T}\}\}$$

In the abstract state 0X the node n_1 is assigned X; this corresponds to two possible Boolean-valued states, one in which n_1 is F and one in which n_1 is T. The nature of this abstraction will be made more precise in later sections.

If one views abstract 'states' s_1 and s_2 as *constraints* or *predicates* on the actual, i.e. Boolean, state of the hardware, then $s_1 \sqsubseteq s_2$ means that every Boolean state that satisfies s_1 also satisfies s_2. We say that s_1 is 'weaker than' s_2. (Strictly speaking, $\sqsubseteq$ is reflexive and we really mean 'no stronger than', but it is common to be somewhat inexact and just say 'weaker than'.) The top value $\top$ represents the unsatisfiable constraint. The *join* operator on pairs of states in the lattice is denoted by '$\sqcup$'.

The theory of symbolic trajectory evaluation can in fact be developed for any complete lattice of states [3]. But this generality is not exploited in mainstream implementations of STE, and so we restrict the presentation in this paper to the simple state lattice above.

2.1.3 Time-dependent Behaviour

To model dynamic behaviour, we represent time by the natural numbers $\mathbb{N}$. A sequence of states that the circuit passes through as it evolves over time is then represented by a function from time to states:

$$\sigma \in \mathbb{N} \to \mathcal{S}$$

Such a function is called a *sequence*. A sequence that does not produce the top state $\top$ just assigns a value in $\{0, 1, \mathrm{X}\}$ to each circuit node at each point in time. For example, $\sigma\ 3\ \mathsf{reset}$ would be the value assigned to the reset node at time 3.

The ordering on states is extended point-wise to sequences in the usual way:

$$\sigma_1 \sqsubseteq \sigma_2 \quad \triangleq \quad \sigma_1(t) \sqsubseteq \sigma_2(t) \text{ for all } t \in \mathbb{N}$$

If $\sigma_1 \sqsubseteq \sigma_2$, then we say that the sequence σ_1 is 'weaker than' the sequence σ_2.

One convenient operation, which will be used later in stating the semantics of STE, is taking the ith suffix of a sequence. The ith suffix of a sequence σ is written σ^i and is defined by

$$\sigma^i\ t \triangleq \sigma\ (t{+}i) \text{ for all } t \in \mathbb{N}.$$

The suffix operation σ^i just shifts the sequence σ forward i points in time, discarding the first i states.

2.1.4 The Next State Function and Trajectories

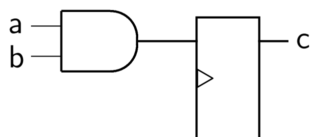
In symbolic trajectory evaluation, the formal model of a circuit c is given by a next-state function Y_c that maps states to states:

$$\mathrm{Y}_c \in \mathcal{S} \rightarrow \mathcal{S}$$

The subscript in Y_c identifies the particular circuit of interest—you can think of 'c' as a constant that names the circuit. This subscript is sometimes omitted when it is clear or doesn't matter which circuit is being discussed.

Intuitively, the next-state function expresses a constraint on the set of possible states into which the circuit may go for any given set of states it might currently be in. Suppose the circuit is in 'abstract' state s, which as we saw above really stands for a set of Boolean states. Then $\mathrm{Y}\ s$ will be the least specified abstract state the circuit can make a transition to. Here, 'least specified' means that if it is consistent with the circuit for a node can take on both 1 and 0 values in the next state, then $\mathrm{Y}(s)$ will assign the value X to that node.

A small example is this simple unit-delay AND-gate:



This circuit has three nodes, which for the purpose of writing down states we order $\mathsf{a} < \mathsf{b} < \mathsf{c}$. A partial tabulation of its Y function is shown below:

$$\begin{array}{rcl} s & = & 1\,1\,1 \quad 1\,1\,0 \quad 1\,0\,1 \cdots\ 1\,1\mathrm{X} \quad 1\,0\,\mathrm{X} \cdots\ \mathrm{X}1\mathrm{X} \quad \mathrm{X}0\mathrm{X} \cdots\ \mathrm{XXX} \\ \mathrm{Y}(s) & = & \mathrm{XX1} \quad \mathrm{XX1} \quad \mathrm{XX0} \cdots\ \mathrm{XX1} \quad \mathrm{XX0} \cdots\ \mathrm{XXX} \quad \mathrm{XX0} \cdots\ \mathrm{XXX} \end{array}$$

Reading from the left, we first see that if the inputs a and b are both 1, then the next state is XX1, regardless of whether c is initially 0 or 1. Hence the output c is 1 in the next state and the inputs a and b are both X (i.e. they can be either Boolean value). In fact, the value of c in the next state doesn't depend on the value of c in the current state, so a little further along in the table we also find $\mathrm{Y}(11\mathrm{X}) = \mathrm{XX1}$.

We also see that if b is 0 in the current state, then the output c is going to be 0 in the next state—regardless of the value of a in the current state. Hence we have Y(X0X) = XX0—we know what the output value will be, even when we have no information about the value on a. Finally, we sometimes have insufficient information to determine the value of the output. If the current state is X1X, for example, then we don't know whether a is going to be 0 or 1—it may be either, and hence Y(10X) = XXX.

A requirement for trajectory evaluation is that the next-state function Y is monotonic. That is, for all states s_1 and s_2

$$s_1 \sqsubseteq s_2 \text{ implies } \mathrm{Y}(s_1) \sqsubseteq \mathrm{Y}(s_2).$$

Implementations of STE must extract a next-state function that has this property from the circuit under analysis. This condition can be met for a wide variety of common circuit design styles, including synchronous systems with both latches and flip-flops, as well as systems with gated clocks.

For a circuit c, we define the set of all its trajectories, $\mathcal{T}(c)$, as follows:

$$\mathcal{T}(c) \triangleq \{\sigma \mid \mathrm{Y}_c(\sigma\ t) \sqsubseteq \sigma\ (t{+}1) \text{ for all } t \in \mathbb{N}\}$$

For a sequence σ to be a trajectory, the result of applying Y_c to any state must be no more specified (with respect to the $\sqsubseteq$ ordering) than the state at the next moment of time. This ensures that σ is consistent with the circuit model Y_c.

2.2 Trajectory Evaluation Logic

One of the keys to the efficiency of STE and its success with datapath circuits is its restricted temporal logic. A *trajectory formula* is a simple linear-time temporal logic formula with the following syntax:

f	:=	n is 0	- node n has value 0
	|	n is 1	- node n has value 1
	|	f_1 and f_2	- conjunction of formulas
	|	$P \to f$	- f is asserted only when P is true
	|	Nf	- f holds in the next time step

where f and g range over formulas, $n \in \mathcal{N}$ ranges over the nodes of the circuit, and P is a propositional formula over Boolean variables (i.e. a Boolean 'function') called a *guard*.

The basic trajectory formulas 'n is 0' and 'n is 1' say that the circuit node n has value 0 or value 1, respectively. The operator and forms the conjunction of trajectory formulas. The trajectory formula $P \to f$ weakens the sub-formula f by

requiring it to be satisfied only when the guard P is true. Finally, $\mathsf{N}f$ says that the trajectory formula f holds in the next point of time.

In essence, a trajectory formula represents a whole *set* of assertions about presence of the Boolean values 0 and 1 on particular circuit nodes. A guard is a propositional formula that may contain Boolean variables, and a trajectory formula $P \rightarrow f$ with a guard P asserts f only for satisfying assignments of values to the Boolean variables in P. So for any trajectory formula, each assignment of values to the variables in its guards gives a (possibly different) assertion about 0s and 1s on certain circuit nodes at particular points in time.

The various guards that occur in a trajectory formula can have variables in common, so this mechanism gives STE the expressive power needed to represent interdependencies among node values. For example, we can associate an arbitrary propositional formula with a node using the construct 'n is P' defined by

$$n \text{ is } P \triangleq P \rightarrow (n \text{ is } 1) \text{ and } \neg P \rightarrow (n \text{ is } 0)$$

We can then specify input-output functions using this construct. For example, we might require that if 'a is x' then 'b is $F[x]$' for some input node a and output node b.

2.2.1 Semantics

Let $\mathcal{V}$ denote the set of Boolean variables that appear in the guards of trajectory formulas and $\mathbb{B} = \{\mathrm{T}, \mathrm{F}\}$ be the Boolean truth-values. Satisfaction is defined with respect to an assignment $\phi \in \mathcal{V} \rightarrow \mathbb{B}$ of Boolean truth-values to the variables that appear in the guards of the formula. Following conventional terminology from logic semantics, we call ϕ a *valuation*. We also write $\phi \models P$ to mean that the propositional formula P is true under the valuation ϕ.

For a given valuation ϕ, we define when a sequence σ satisfies a trajectory formula recursively as follows:

$$\begin{array}{lcl}
\phi, \sigma \models n \text{ is } 0 & \triangleq & \begin{cases} \sigma(0) = \top, \text{ or} \\ \sigma(0) \in \mathcal{N} \rightarrow \mathcal{D} \text{ and } \sigma\ 0\ n = 0 \end{cases} \\
\phi, \sigma \models n \text{ is } 1 & \triangleq & \begin{cases} \sigma(0) = \top, \text{ or} \\ \sigma(0) \in \mathcal{N} \rightarrow \mathcal{D} \text{ and } \sigma\ 0\ n = 1 \end{cases} \\
\phi, \sigma \models f \text{ and } g & \triangleq & \phi, \sigma \models f \text{ and } \phi, \sigma \models g \\
\phi, \sigma \models P \rightarrow f & \triangleq & \phi \models P \text{ implies that } \phi, \sigma \models f \\
\phi, \sigma \models \mathsf{N}f & \triangleq & \phi, \sigma^1 \models f
\end{array}$$

Note that the same valuation ϕ applies to all the guards that appear in a trajectory formula—so the scope of any Boolean variable is the entire formula. The valuation also does not depend on time.

2.2.2 Trajectory Assertions

Circuit correctness in symbolic trajectory evaluation is stated with *trajectory assertions* of the form $A \Rightarrow C$, where A and C are trajectory formulas. The intuition is that the *antecedent* A provides stimuli to circuit nodes and the *consequent* C specifies the values expected on circuit nodes as a response.

A trajectory assertion is true for a given assignment ϕ of Boolean values to the variables in its guards exactly when every trajectory of the circuit that satisfies the antecedent also satisfies the consequent. For a given circuit c, we define $\phi \models A \Rightarrow C$ to mean that for all $\sigma \in \mathcal{T}(c)$, if $\phi, \sigma \models A$ then $\phi, \sigma \models C$. The notation $\models A \Rightarrow C$ means that $\phi \models A \Rightarrow C$ holds for all ϕ.

2.3 Model Checking Trajectory Assertions

The key feature of this logic is that for any trajectory formula f and assignment ϕ, there exists a unique weakest sequence that satisfies f. This sequence is called the *defining sequence* for f and is written $[f]^\phi$. It is defined recursively as follows:

$$
\begin{array}{lll}
[m \text{ is } 0]^\phi \; t\, n & \triangleq & 0 \text{ if } m{=}n \text{ and } t{=}0, \text{ otherwise X} \\
[m \text{ is } 1]^\phi \; t\, n & \triangleq & 1 \text{ if } m{=}n \text{ and } t{=}0, \text{ otherwise X} \\
[f \text{ and } g]^\phi \; t & \triangleq & ([f]^\phi\, t) \sqcup ([g]^\phi\, t) \\
[P \rightarrow f]^\phi \; t\, n & \triangleq & [f]^\phi\, t\, n \text{ if } \phi \models P, \text{ otherwise X} \\
[\mathsf{N} f]^\phi \; t\, n & \triangleq & [f]^\phi\, (t{-}1)\, n \text{ if } t{\neq}0, \text{ otherwise X}
\end{array}
$$

The crucial property enjoyed by this definition is that $[f]^\phi$ is the unique weakest sequence that satisfies f for the given ϕ. That is, for any ϕ and σ, $\phi, \sigma \models f$ if and only if $[f]^\phi \sqsubseteq \sigma$.

The algorithm for STE is also concerned with the weakest *trajectory* that satisfies a particular formula. This is the *defining trajectory* for a formula, written $[\![f]\!]^\phi$. It is defined by the following recursive calculation:

$$
\begin{array}{lll}
[\![f]\!]^\phi\; 0 & \triangleq & [f]^\phi\; 0 \\
[\![f]\!]^\phi\; (t{+}1) & \triangleq & [f]^\phi\; (t{+}1) \sqcup \mathrm{Y}_c([\![f]\!]^\phi\; t)
\end{array}
$$

The defining trajectory of a formula f is its defining sequence with the added constraints on state transitions imposed by the circuit, as modelled by the next-state function Y_c. It can be shown that $[\![f]\!]^\phi$ is the unique weakest trajectory that satisfies f. That is, for any ϕ and σ, we have that $\sigma \in \mathcal{T}(c)$ and $\phi, \sigma \models f$ if and only if $[\![f]\!]^\phi \sqsubseteq \sigma$.

The fundamental theorem of trajectory evaluation [3] follows immediately from the previously-stated properties of $[f]^\phi$ and $[\![f]\!]^\phi$.

Theorem 1 *For any valuation ϕ and trajectory assertion $A \Rightarrow C$, $\phi \models A \Rightarrow C$ exactly when $[C]^\phi \sqsubseteq [\![A]\!]^\phi$.*

The intuition is that the sequence characterising the consequent must be 'included in' the weakest sequence satisfying the antecedent that is also consistent with the circuit.

This theorem gives a model-checking algorithm for trajectory assertions: to see if $\phi \models A \Rightarrow C$ holds for a given ϕ, just compute $[C]^\phi$ and $[\![A]\!]^\phi$ and compare them point-wise for every circuit node and point in time. This works because both A and C will have only a finite number of nested next-time operators N, and so only finite initial segments of the defining trajectory and defining sequence need to be calculated and compared.

In practice, the defining trajectory of A and the defining sequence of C are computed iteratively, and each state is checked against the ordering requirement as it is generated. Each state of the defining trajectory is computed from the previous state by simulation of a netlist description of the circuit, in which circuit nodes take on values in $\{0, 1, \mathrm{X}\}$.

2.3.1 Symbolic Trajectory Evaluation

The model checking algorithm just sketched requires ϕ to be supplied; given a specific assignment ϕ of values to Boolean variables in the guards of a formula, we can calculate and point-wise compare $[C]^\phi$ and $[\![A]\!]^\phi$. But much of the debugging power of STE comes from the key observation that it is not necessary to supply ϕ in advance. Instead, the comparison can be computed 'symbolically' to give a *constraint* on ϕ. This constraint is called a *residual* and represents precisely the conditions under which the property $A \Rightarrow C$ is true of the circuit.

This symbolic version of the model checking algorithm works as follows. At the level of basic data values in $\{0, 1, \mathrm{X}\}$, the required computation is to show that

$$[C]^\phi\ t\ n \leq [\![A]\!]^\phi\ t\ n \qquad (1)$$

for all $t \geq 0$ and $n \in \mathcal{N}$. For each circuit node at each relevant point in time, we compare the data values expected by the consequent to those given by the circuit and antecedent. To make this comparison 'symbolic', we use a pair of BDDs to encode functions from ϕ to data values in $\mathcal{D}$. This is the so-called 'dual-rail' encoding employed STE implementations [11]. We also extend the simulation algorithm to a symbolic version, in which data values are these pairs of BDDs. The model checking algorithm then compares symbolic states, resulting in the residual.

2.4 The STE Deductive System

STE has a sound and complete deductive system for proving trajectory formulas [12, 13], which have been implemented as a group of inference rules in the Forte theorem prover. The complete set of rules is as follows.

Reflexivity. $\models A \Rightarrow A$ holds for any trajectory formula A.

Time Shift. For any trajectory formulas A and C, if $\models A \Rightarrow C$ then $\models \mathsf{N}A \Rightarrow \mathsf{N}C$.

Antecedent Strengthening. For any trajectory formulas A and C, if $\models A \Rightarrow C$ then for any trajectory formula A' for which $[A]^\phi \sqsubseteq [A']^\phi$ for all ϕ, we have $\models A' \Rightarrow C$.

Consequent Weakening. For any trajectory formulas A and C, if $\models A \Rightarrow C$ then for any trajectory formula C' for which $[C']^\phi \sqsubseteq [C]^\phi$ for all ϕ, we have $\models A \Rightarrow C'$.

Conjunction. For any trajectory formulas A_1, A_2, C_1, and C_2, if $\models A_1 \Rightarrow C_1$ and $\models A_2 \Rightarrow C_2$, then $\models A_1 \ \mathsf{and} \ A_2 \Rightarrow C_1 \ \mathsf{and} \ C_2$.

Transitivity. For any trajectory formulas A, B, and C, if $\models A \Rightarrow B$ and $\models B \Rightarrow C$, then $\models A \Rightarrow C$.

Substitution. For any trajectory formulas A and C, if $\models A \Rightarrow C$, then $\models A[Ps/xs] \Rightarrow C[Ps/xs]$ for any substitution of formulas Ps for Boolean variables xs.

The main purpose of these rules is to combine individual STE model-checking results together [14] to derive correctness results that are infeasible to model-check directly. The inference rules can also be used to transform trajectory formulas to increase model checking efficiency [15].

Acknowledgements

Koen Claessen, John Harrison, Ed Smith, and Jan-Willem Roorda kindly read earlier drafts of this report and provided many helpful comments. Ashish Darbari is supported by a donation from Intel Corporation for research into *Problem Reduction for Combined Model-Checking and Theorem Proving.*

References

[1] P. R. Halmos, *Naive Set Theory*, Springer-Verlag, 1987.

[2] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[3] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, March 1995.

[4] A. Jain, *Formal Hardware Verification by Symbolic Trajectory Evaluation*, Ph.D. thesis, Carnegie Mellon University, August 1997.

[5] J. Yang and C.-J. H. Seger, "Introduction to generalized symbolic trajectory evaluation," in *Proceedings of 2001 IEEE International Conference on Computer Design*. 2001, pp. 360–365, IEEE.

[6] R. E. Bryant, "A methodology for hwardware verification based on logic simulation," *Journal of the ACM*, vol. 38, no. 2, pp. 299–328, April 1991.

[7] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[8] R. E. Bryant, D. E. Beatty, and C.-J. H. Seger, "Formal hardware verification by symbolic ternary trajectory evaluation," in *ACM/IEEE Design Automation Conference*, 1991, pp. 297–402.

[9] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[10] C.-T. Chou, "The mathematical foundation of symbolic trajectory evaluation," in *Computer Aided Verification: 11th International Conference, Trento, Italy, July 6-10 1999: Proceedings*, N. Halbwachs and D. Peled, Eds. 1999, vol. 1633 of *Lecture Notes in Computer Science*, Springer-Verlag.

[11] C.-J. H. Seger, "Voss — a formal hardware verification system: User's guide," Tech. Rep. TR-93-45, University of Brtish Columbia Department of Computer Science, December 1993.

[12] S. Hazelhurst and C.-J. H. Seger, "A simple theorem prover based on symbolic trajectory evaluation and BDDs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, apr 1995.

[13] Z. Zhu and C.-J. Seger, "The completeness of a hardware inference system," in *Computer Aided Verification*, 1994, pp. 286–298.

[14] S. Hazelhurst and C.-J. H. Seger, "Symbolic trajectory evaluation," in *Formal Hardware Verification*, T. Kropf, Ed., chapter 1, pp. 3–78. Springer-Verlag, 1997.

[15] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of Boolean constraints," in *ACM/IEEE Design Automation Conference*, July 1998.