A New SAT-based Algorithm for Symbolic Trajectory Evaluation

Jan-Willem Roorda and Koen Claessen
{jwr,koen}@cs.chalmers.se

Chalmers University of Technology, Sweden

Abstract. We present a new SAT-based algorithm for Symbolic Trajectory Evaluation (STE), and compare it to more established SAT-based techniques for STE.

1 Introduction

Symbolic Trajectory Evaluation (STE) [7] is a high-performance simulation-based model checking technique. It combines three-valued simulation (using the standard values 0 and 1 together with the extra value X, "don't know") with symbolic simulation (using symbolic expressions to drive inputs). STE has been extremely successful in verifying properties of circuits containing large data paths (such as memories, fifos, floating point units) that are beyond the reach of traditional symbolic model checking [1, 6, 5].

STE specifications are written in a restricted temporal language, where assertions are of the form $A \Longrightarrow C$; the *antecedent* A drives the simulation, and the *consequent* C expresses the conditions that should result. In the assertion variables are taken from a set of Boolean symbolic variables V.

In STE, two abstractions are used: (1) the value X can be used to abstract from a specific Boolean value of a circuit node, (2) information is only propagated forwards through the circuit and through time. A *trajectory* is a sequence of node assignments over time that meets the constraints of the circuit taking these abstractions into account. An STE-assertion $A \Longrightarrow C$ holds if each trajectory that satisfies A also satisfies C.

STE Model Checking All current implementations of STE use symbolic simulation to compute a representation of the so-called *weakest* trajectory that satisfies the antecedent A. While computing this representation, it is checked if the trajectory also satisfies the consequent C. Such a weakest trajectory can be represented by means of BDDs and a dual-rail encoding. A pleasant property of STE is that the number of variables occurring in these BDDs only depends on the number of variables in the STE assertion, not on the size of the circuit.

An alternative way of implementing STE is to use SAT. Bjesse et al. [3] and Singerman et al. [9] independently implemented SAT-based STE by using a simulator that works on non-canonical Boolean expressions instead of BDDs. The STE symbolic simulator calculates a symbolic expression for the weakest trajectory satisfying the antecedent. After simulation, the propositional formula expressing that weakest trajectory satisfies the consequent is fed to a SAT-solver.

Bjesse et al. used SAT-based STE for bug finding for a design of an Alpha microprocessor. The authors report that SAT-based STE enabled them to find bugs as deep as with Bounded Model Checking, but with negligible run-times. Singerman et al. showed how SAT-based STE can be used for bug finding in *Generalized Symbolic Trajectory Evaluation* (GSTE). This bug finding method is called satGSTE. GSTE [10] is a generalization of STE that can verify properties over infinite time intervals. The core of the satGSTE algorithm is a SAT-based algorithm for (non-generalized) STE, as described above. At Intel, satGSTE is used for debugging and refining GSTE assertion graphs, thereby improving user productivity.

Contributions We have developed an alternative, more efficient, method of verifying STE properties using SAT. The idea is that, instead of simulating the circuit and creating a symbolic expression for the *weakest* trajectory satisfying the antecedent but not the consequent, our algorithm generates a constraint problem that represents *all* trajectories satisfying the antecedent and not the consequent. We argue that this approach is much better suited for use with a SAT-solver.

A second contribution is an alternative STE semantics, that is closely related to our algorithm, and more faithfully describes the behaviour of existing STE algorithms.

In the following, we present our STE semantics, and show how to convert the semantic definitions directly into *primitive abstract constraints*. We then show how to implement these primitive abstract constraints using a SAT-solver, and compare running times on some benchmarks with other SAT-based approaches.

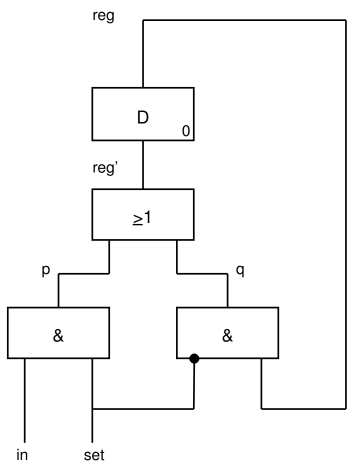

Fig. 1. Gate-level model of a memory cell circuit

$$\begin{aligned} \mathsf{p} &= \mathsf{set}\ \text{AND}\ \mathsf{in} \\ \mathsf{q} &= \neg\mathsf{set}\ \text{AND}\ \mathsf{reg} \\ \mathsf{reg}' &= \mathsf{p}\ \text{OR}\ \mathsf{q} \end{aligned}$$

Fig. 2. Netlist of the circuit

2 Preliminaries

Circuits A circuit is modeled by a set of node names $\mathcal{N}$ connected by logical gates and delay elements. $\mathcal{S} \subseteq \mathcal{N}$ is the set of state holding nodes, used to model delay elements. It is assumed that for every node n in $\mathcal{S}$, there is a node n' in $\mathcal{N}$ that models the value of that node in the next state.

It is common to describe a circuit in the form of a *netlist*. Here, a netlist is an acyclic list of definitions describing the relations between the values of the nodes. Consider for example the gate-level model of a memory cell circuit in Fig. 1. The netlist of this circuit is given in Fig. 2. Inverters are not modeled explicitly in our netlists, instead they occur implicitly for each mention of the negation operator $\neg$ on the inputs of the gates. Delay elements are not mentioned explicitly in the netlist either. Instead, for a register with output node n in the circuit, the input of the delay element is node n' which is mentioned in the netlist. So, from the netlist in Fig. 2 it can be derived that the node reg is the output of a delay element with input reg'. The netlists used here do not contain the initial values of delay elements. They are not needed as the STE abstraction assumes that the initial states of delay elements are unknown. For simplicity, we only allow AND-gates and OR-gates in netlists. It is, however, straightforward to extend this notion of netlists to include more operations.

Values In STE, we can abstract away from specific Boolean values of a node taken from the set $\mathbb{B} = \{0, 1\}$, by using the value X, which stands for *unknown*. The set of signal values is denoted $\mathbb{V} = \{0, 1, \mathsf{X}\}$. On this set an *information-ordering* $\leq$ is introduced. The unknown value X contains the least information, so $\mathsf{X} \leq 0$ and $\mathsf{X} \leq 1$, while 0 and 1 are incomparable. If $v \leq w$ it is said that v is *weaker* than w.

A *circuit state*, written $s : \mathbf{State}$, is a function from $\mathcal{N}$ to $\mathbb{V}$, assigning a value from $\mathbb{V}$ to each node in the circuit. A *sequence* $\sigma : \mathbb{N} \to \mathbf{State}$ is a function from a point in time to a circuit state, describing the behaviour of a circuit over time. The set of all sequences σ is written $\mathbf{Seq}$.

Trajectory Evaluation Logic STE assertions have the form $A \Longrightarrow C$. Here A and C are formulas in *Trajectory Evaluation Logic* (TEL). The only variables in the logic are time-independent Boolean variables taken from the set V of *symbolic variables*. The language is given by the following grammar:

$$f \ ::= \ n \,\mathbf{is}\, 0 \ | \ n \,\mathbf{is}\, 1 \ | \ f_1 \ \mathbf{and} \ f_2 \ | \ P \to f \ | \ \mathbf{N} f$$

where $n \in \mathcal{N}$ and P is a Boolean propositional formula over the set of symbolic variables V. The operator **is** is used to make a statement about the Boolean value of a particular node in the circuit, **and** is conjunction, $\to$ is used to make conditional statements, and $\mathbf{N}$ is the next time operator. Note that symbolic variables only occur in the Boolean propositional expressions on the left-hand side of an implication. The notation n **is** P, where P is a Boolean symbolic expression over the set of symbolic variables V, is used to abbreviate the formula: $(\neg P \to n \,\mathbf{is}\, 0)$ **and** $(P \to n \,\mathbf{is}\, 1)$.

The meaning of a TEL formula is defined by a satisfaction relation that relates valuations of the symbolic variables and sequences to TEL formulas. Here, the following notation is used: The time shifting operator σ^1 is defined by $\sigma^1(t)(n) = \sigma(t+1)(n)$. Standard propositional satisfiability is denoted by $\models_{\text{Prop}}$. Satisfaction of a trajectory evaluation logic formula f, by a sequence $\sigma \in \mathbf{Seq}$, and a valuation $\phi : V \to \mathbb{B}$ (written $\phi, \sigma \models f$) is defined by

$$\begin{array}{lcl}
\phi, \sigma \models n \,\mathbf{is}\, b & \equiv & \sigma(0)(n) = b \ \ , \ \ b \in \{0,1\} \\
\phi, \sigma \models f_1 \ \mathbf{and} \ f_2 & \equiv & \phi, \sigma \models f_1 \ \text{ and } \ \phi, \sigma \models f_2 \\
\phi, \sigma \models P \to f & \equiv & \phi \models_{\text{Prop}} P \ \text{ implies } \ \phi, \sigma \models f \\
\phi, \sigma \models \mathbf{N} f & \equiv & \phi, \sigma^1 \models f
\end{array}$$

n	$s_1(n)$
in	1
set	1
p	1
q	0
reg′	1
reg	X

n	$s_2(n)$
p	0
other	X

Fig. 3. Example states

3 Stability Semantics

In STE model-checking two abstractions are used: (1) the value X can be used to abstract from a specific Boolean value of a circuit node, (2) information is only propagated forwards through the circuit (i.e. from inputs to outputs of gates) and through time (i.e. from time t to time $t+1$). Given a circuit *c*, a *trajectory* is a sequence that meets the constraints of the circuit *c*, taking these abstractions into account. An STE-assertion $A \Longrightarrow C$ holds if each trajectory that satisfies A also satisfies C.

For instance, for the memory cell given in Fig. 1, consider the assertion: $\mathsf{p}\ \mathbf{is}\ 1 \Longrightarrow \mathsf{reg}'\ \mathbf{is}\ 1$. The antecedent specifies the value 1 for node p, so each trajectory satisfying the antecedent should give node p value 1. As node reg' is the output of an OR-gate with input node p, the node reg' is, by forwards propagation, required to have value 1 in each such trajectory. Therefore the assertion is true in STE.

The assertion $\mathsf{p}\ \mathbf{is}\ 1 \Longrightarrow \mathsf{set}\ \mathbf{is}\ 1$ is, however, not true. Node set is the input to an AND-gate with output node p. But, as there is no *backwards* propagation of information in STE, a trajectory for the memory cell is allowed to give node p value 1 while giving node set value X at the same time point.

Also the assertion $(\mathsf{in}\ \mathbf{is}\ 1)\ \mathbf{and}\ (\mathsf{reg}\ \mathbf{is}\ 1) \Longrightarrow (\mathsf{reg}'\ \mathbf{is}\ 1)$ is not true in STE. Although for each *Boolean* value of node set, node reg' is, by forwards propagation, required to have value 1, the sequence giving both node set and reg' value X is a trajectory that satisfies the antecedent but not the consequent.

Semantics Below we define a new semantics for STE. The reason we give a new semantics here is that the "classic" semantics of STE [7] cannot deal with combinatorial assertions. That is, it cannot deal with assertions that express a relation between circuit nodes at the same time-point. Because our algorithm (just as existing algorithms like the STE algorithm in Intel's in-house verification toolkit Forte [4]) can prove such properties, we needed a new semantics to prove our algorithm sound and complete.

Stable State Sets To model this behaviour of STE, we propose to use *stable state sets*, written $F : \mathbb{P}(\mathbf{State})$ as circuit-models. The idea is that a (three-valued) state s is in the stable state set F_c of a circuit c if no more information about the circuit state at the *same* point in time can be derived by propagating the information in a *forwards* fashion. Later, we define trajectories in terms of stables state sets.

Example 1. In this example (and following examples), a state $s : \{p, q, r\} \to \mathbb{V}$ is written as a vector $s(p)s(q)s(r)$. Consider the circuit consisting of a single AND-gate with inputs p and q, and output r. The stable state set of this circuit is

$$\begin{aligned} F_c &= \{\ s \mid \text{if } s(\mathsf{p}) = s(\mathsf{q}) = 1 \text{ then } s(\mathsf{r}) = 1,\ \text{ if } s(\mathsf{p}) = 0 \text{ or } s(\mathsf{q}) = 0 \text{ then } s(\mathsf{r}) = 0\ \} \\ &= \{000, 010, 0\mathsf{X}0, 100, 111, 1\mathsf{X}0, 1\mathsf{X}1, 1\mathsf{X}\mathsf{X}, \mathsf{X}00, \mathsf{X}10, \mathsf{X}11, \mathsf{X}1\mathsf{X}, \mathsf{X}\mathsf{X}0, \mathsf{X}\mathsf{X}1, \mathsf{X}\mathsf{X}\mathsf{X}\} \end{aligned}$$

The state $0\mathsf{X}0$ is in the stable state set, because if $\mathsf{p} = 0$ then $\mathsf{r} = 0$, but no new information about q can be derived. Also, $\mathsf{XX}1$ is in the stable state set; the reason is that from $\mathsf{r} = 1$, we cannot derive information about p or q by means of forwards propagation. The state $11\mathsf{X}$ is not in the stable state set of the circuit; when $\mathsf{p} = 1$ and $\mathsf{q} = 1$, forwards propagation requires that also the output has value 1. □

Given the netlist of a circuit, the circuit's stable state set is constructed by taking the intersection of all stable state sets belonging to each of the gates. The stable state sets of AND- and OR-gates with inputs p and q and output r are written $F_{\text{AND}}(p, q, r)$ and $F_{\text{OR}}(p, q, r)$, respectively. The definition of $F_{\text{AND}}(p, q, r)$ is given in Example 1. The set $F_{\text{OR}}(p, q, r)$ is defined similarly. Here, note that the stable state set of a gate is a set of states of the whole circuit and not a set of states of only the in- and outputs of the gate.

Example 2. The stable state set for the memory cell from Fig. 1 is given by:

$$F_c = F_{\text{AND}}(\mathsf{set}, \mathsf{in}, \mathsf{p})\ \cap\ F_{\text{AND}}(\neg\mathsf{set}, \mathsf{reg}, \mathsf{q})\ \cap\ F_{\text{OR}}(\mathsf{p}, \mathsf{q}, \mathsf{reg}')$$

Consider the states s_1, s_2 given in Fig. 3. State s_1 is in the stable state set F_c as all node assignments are consistent and no new information can be derived. State s_2 given in Fig. 3 is also in the stable state set of the memory cell as from the node-assignment $p = 0$ no information can be derived by forwards propagation of information. $\square$

Trajectories A trajectory is a sequence in which no more information can be derived by forwards propagation of information. Recall that for every delay element with output n the input to the delay element is called n'. Therefore, in a trajectory, the value of node n' at time t should be propagated to node n at time $t + 1$.

So, a sequence σ is a trajectory if for each time point $t \in \mathbb{N}$: (1) the state $\sigma(t)$ is a stable state, and (2) for each state holding node $n \in \mathcal{S}$, the value of node n at time $t + 1$ contains at least the same information as the value of node n' at time t. More formally, the set of trajectories of a circuit c, written $F_c^{\rightarrow}$: **Seq**, is defined by:

$$F_c^{\rightarrow} \;=\; \{\, \sigma \mid \forall t \in \mathbb{N} \,.\, \sigma(t) \in F_c,\;\; \forall t \in \mathbb{N} \,.\, \forall n \in \mathcal{S} \,.\, \sigma(t)(n') \leq \sigma(t+1)(n) \,\}$$

Stable Semantics of STE Using the definition of trajectories of a circuit, we can now define the semantics of an STE assertion. A circuit c *satisfies* a trajectory assertion $A \Longrightarrow C$, written $c \models_{\rightarrow} A \Longrightarrow C$ iff for every valuation $\phi \in V \to \mathbb{B}$ of the symbolic variables, and for every trajectory τ of c, it holds that:

$$\phi, \tau \models A \quad \Rightarrow \quad \phi, \tau \models C.$$

Counter Examples A valuation ϕ together with a trajectory τ that satisfies A but not C form a counter example of the STE assertion. Because any given STE assertion only refers to a finite number of points in time, only a finite part of the trajectory τ contains interesting information about the counter example. We call the *depth d* of an assertion the maximum number of nested occurrences of the next time operator $\mathbf{N}$. In order to construct a counter example for an assertion of depth d, it is enough to only consider the first d time points of the trajectory. We will use this fact in the next section.

4 A Constraint-Based Algorithm for STE

In this section, we describe how an STE assertion can be checked using a constraint solver that can solve sets of constraints built-up from a small set of *primitive abstract constraints* with a well-defined meaning. In the next section, we show how to concretely represent each of these primitive abstract constraints as a set of clauses in a SAT solver.

Constraints A *constraint* $S \in \mathsf{Constraint}(D)$ on a domain D is a syntactical object that restricts the elements of D to the set of *solutions* of the constraint. The semantics of constraints is given by the function $\mathsf{sol} : \mathsf{Constraint}(D) \to \mathbb{P}(D)$, yielding all solutions of a given constraint. Constraints can be combined by the conjunction operator &. The solutions of a conjunction of two constraints is the intersection of their sets of solutions, that is: $\mathsf{sol}(S_1 \;\&\; S_2) = \mathsf{sol}(S_1) \cap \mathsf{sol}(S_2)$.

In the following, we present a constraint-based algorithm for STE. The idea is to translate a circuit c and an STE assertion $A \Longrightarrow C$ into a constraint S, such that the STE assertion holds for the circuit if and only if the constraint S has no solutions. Each solution to S represents a counter example, a valuation ϕ and a trajectory τ that together satisfy A but not C.

Domain The solution domain D of our constraints consists of pairs (ϕ, σ) of valuations and sequences. For an STE assertion of depth d, we need only to consider the first d points in time. Therefore, the sequence part of a solution (ϕ, σ) is a function from time points $\{0, \ldots, d\}$ to states.

Given a circuit c and an assertion $A \Longrightarrow C$, the final constraint for the STE problem, written $\mathsf{CEX}(c \models A \Longrightarrow C)$, consists of 3 parts: (1) constraints that restrict the first d time points of the sequences considered to be the first d time points of trajectories of the circuit c, (2) constraints that restrict the sequences and valuations considered to satisfy the antecedent A, and (3) constraints that restrict the sequences and valuations considered to *not* satisfy the consequent C. Thus, if we find a solution that satisfies all three parts, we have found a counter example to the STE assertion. If we show that no such solution exists, we have shown that the STE assertion holds.

Trajectory Constraint Given a circuit c with stable state set F_c, we denote the constraint that restricts the first d time steps of the solutions to be trajectories of c by $\mathsf{TRAJ}(F_c, d)$. It consists of *stable state constraints*, denoted $\mathsf{STABLE}(F_c, t)$, that restrict each point in time t to be a stable state w.r.t. F_c, and of *transition constraints*, denoted $\mathsf{TRANS}(t, t+1)$, that connect the state holding nodes for each point in time t to the next point in time $t+1$:

$$\begin{aligned} \mathsf{TRAJ}(F_c, d) = \quad & \mathsf{STABLE}(F_c, 0) \ \& \ \ldots \ \& \ \mathsf{STABLE}(F_c, d) \\ \& \ & \mathsf{TRANS}(0,1) \ \& \ \ldots \ \& \ \mathsf{TRANS}(d-1, d) \end{aligned}$$

For a given STE assertion of depth d, only the first d points in time of a trajectory are interesting, and thus we only create constraints for the first d steps of the constraint.

The definition of the constraint $\mathsf{STABLE}(F_c, t)$ makes use of the primitive abstract constraints for the AND- and OR-gates, denoted $\mathsf{AND}(p_t, q_t, r_t)$ and $\mathsf{OR}(p_t, q_t, r_t)$. Here the notation n_t refers to the value of node n at time point t. We show how to concretely implement these constraints in the next section. For now, it is only important to know that the solutions to the constraints are exactly the ones allowed by their stable state sets. For example, for the AND-gate constraint it holds:

$$\mathsf{sol}(\mathsf{AND}(p_t, q_t, r_t)) \ = \ \{(\phi, \sigma) | \ \sigma(t) \ \in \ F_{\text{AND}}(p, q, r) \ \}$$

To build the constraint $\mathsf{STABLE}(F_c, t)$ for the stable state of the circuit, we simply follow the structure of the netlist and conjoin the constraints for each gate together.

Example 3. The stable state constraint for the memory cell is given by:

$$\mathsf{AND}(\mathsf{set}_t, \mathsf{in}_t, \mathsf{p}_t) \ \& \ \mathsf{AND}(\neg\mathsf{set}_t, \mathsf{reg}_t, \mathsf{q}_t) \ \& \ \mathsf{OR}(\mathsf{p}_t, \mathsf{q}_t, \mathsf{reg'}_t)$$

□

For a given point in time t, and a circuit c, the transition constraint $\mathsf{TRANS}(t, t+1)$ is built up from primitive abstract constraints of the form $\mathsf{LT}(n_{t_1} \leq m_{t_2})$. The constraint $\mathsf{LT}(n_{t_1} \leq m_{t_2})$ demands that the value of node n at time t_1 is weaker than the value of node m at time t_2. Here, we require:

$$\mathsf{sol}(\mathsf{LT}(n_{t_1} \leq m_{t_2})) \ = \ \{ \ (\phi, \sigma) \mid \sigma(t_1)(n) \ \leq \ \sigma(t_2)(m) \ \}$$

The definition of the constraint $\mathsf{TRANS}(t, t+1)$ then becomes:

$$\mathsf{TRANS}(t, t+1) \;=\; \&_{n \in \mathcal{S}}\ \mathsf{LT}(n'_t \le n_{t+1})$$

Example 4. For the memory cell, $\mathsf{TRAJ}(F_c, 2)$ is given by:

$$\begin{aligned} & \mathsf{AND}(\mathsf{set}_0, \mathsf{in}_0, \mathsf{p}_0) \;\&\; \mathsf{AND}(\neg\mathsf{set}_0, \mathsf{reg}_0, \mathsf{q}_0) \;\&\; \mathsf{OR}(\mathsf{p}_0, \mathsf{q}_0, \mathsf{reg'}_0) \\ \& \; & \mathsf{AND}(\mathsf{set}_1, \mathsf{in}_1, \mathsf{p}_1) \;\&\; \mathsf{AND}(\neg\mathsf{set}_1, \mathsf{reg}_1, \mathsf{q}_1) \;\&\; \mathsf{OR}(\mathsf{p}_1, \mathsf{q}_1, \mathsf{reg'}_1) \\ \& \; & \mathsf{LT}(\mathsf{reg'}_0 \le \mathsf{reg}_1) \end{aligned}$$

□

Proposition 1. *For any circuit c, it holds that:*

$$\mathsf{sol}(\mathsf{TRAJ}(F_c, d)) = \{(\phi, \tau \restriction \{0, 1, .., d\}) \mid \tau \in F_c^{\rightarrow}\}.$$

Antecedent Constraint In order to build the constraint for the antecedent, we need to define the concept of *defining formula*. Given an antecedent A, a node name n, a boolean value $b \in \mathbb{B}$, and a time point t, we can construct a propositional formula that is true exactly when A requires the node n to have value b at time point t. This formula is called the *defining formula*, and is denoted by $\langle A\rangle(t)(n = b)$.

Example 5. If the antecedent A is defined as $(a \wedge b) \rightarrow \mathsf{in}\ \mathbf{is}\ 0$, then $\langle A\rangle(0)(\mathsf{in} = 0)$ is the formula $a \wedge b$, since only when $a \wedge b$ holds, does A require the node in to be 0. However, $\langle A\rangle(0)(\mathsf{in} = 1)$ is the false formula 0, since A never requires the node in to be 1. □

The *defining formula* is defined recursively as follows:

$$\begin{aligned} \langle m\ \mathbf{is}\ b'\rangle(t)(n = b) &= \begin{cases} 1, \text{ if } m = n, b' = b \text{ and } t = 0 \\ 0, \text{ otherwise} \end{cases} \\ \langle f_1\ \mathbf{and}\ f_2\rangle(t)(n = b) &= \langle f_1\rangle(t)(n = b) \vee \langle f_2\rangle(t)(n = b) \\ \langle P \rightarrow f\rangle(t)(n = b) &= P \wedge \langle f\rangle(t)(n = b) \\ \langle \mathbf{N} f\rangle(t)(n = b) &= \begin{cases} \langle f\rangle(t-1)(n = b), & \text{if } t > 0 \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

Note that for an antecedent of the form f_1 **and** f_2 to require that a node has a value, it is enough that one of the formulas f_1 or f_2 requires this.

The third primitive abstract constraint is called an *implication constraint*, and given a propositional formula P, a node n, time point t, and a boolean value b, is written $\mathsf{IMPLIES}(\ P \rightarrow (n_t = b)\ )$. The meaning of this constraint is required to be:

$$\mathsf{sol}(\mathsf{IMPLIES}(\ P \rightarrow (n_t = b)\ )) = \{(\phi, \sigma) \mid \text{if } \phi \models P \text{ then } \sigma(t)(n) = b\ \}$$

Lastly, the constraint for the antecedent, written $\mathsf{SAT}(A)$, is defined by:

$$\mathsf{SAT}(A) \;=\; \&_{0 \le t \le d}.\ \&_{n \in \mathcal{N}}.\ \&_{b \in \mathbb{B}}.\mathsf{IMPLIES}(\ \langle A\rangle(t)(n = b) \rightarrow (n_t = b)\ )$$

In other words, we take the conjunction of all requirements that the antecedent A might have on any node n at any time t with any value b.

Example 6. For the TEL formula $A = (\mathsf{in}\ \mathbf{is}\ a)$:

$$\mathsf{SAT}(A) = \mathsf{IMPLIES}(\ \neg a \to (\mathsf{in}_0 = 0)\ )\ \ \&\ \ \mathsf{IMPLIES}(\ a \to (\mathsf{in}_0 = 1)\ )$$

Proposition 2. *For every TEL-formula A of depth d:*

$$\mathsf{sol}(\mathsf{SAT}(A)) = \{(\phi, \sigma \restriction \{0, 1, .., d\}) \mid \phi, \sigma \models A\}.$$

Consequent Constraint For the consequent, we should add a constraint that *negates* the requirements of the consequent on the values of the circuit nodes. In order to do so, we introduce a fresh symbolic variable k_t^n for each node[1] $n \in \mathcal{N}$ and time point $t \in \{0, \ldots, d\}$. We force the variable k_t^n to have value 0 if node n at time t satisfies the requirements of the consequent C. There are three cases when this happens: (1) C requires node n at time t to have value 1 and it has indeed value 1. (2) C requires node n at time t to have value 0 and it has indeed value 0. (3) C has no requirements on node n at time t. Finally, a constraint is introduced that requires that at least one of the k_t^n has value 1. This constrains the set of solutions to contain only solutions where at least one of the requirements of C is not fulfilled.

For the definition of negation of the consequent, two more primitive abstract implication constraints are introduced:

$$\begin{array}{l} \mathsf{IMPLIES}(\ \ (P \text{ and } (n_t = b)) \to k_t^n = 0\ ) \\ \mathsf{IMPLIES}(\ P \to k_t^n = 0\ ) \end{array}$$

The meaning of these constraints is given by:

$$\begin{array}{c} \mathsf{sol}(\mathsf{IMPLIES}(\ (P \text{ and } (n_t = b)) \to k_t^n = 0\ )) \\ = \{(\phi, \sigma) \mid \text{if } \phi \models P \text{ and } \sigma(t)(n) = b \text{ then } \phi(k_t^n) = 0\} \end{array}$$

$$\mathsf{sol}(\mathsf{IMPLIES}(\ \ P \to k_t^n = 0\ )) = \{(\phi, \sigma) \mid \text{if } \phi \models P \text{ then } \phi(k_t^n) = 0\}$$

Furthermore, a primitive abstract constraint that demands that at least one of the k_t^n has value 1, written $\mathsf{EXISTS}(k_t^n = 1)$ is needed. The meaning of this constraint is given by:

$$\begin{array}{c} \mathsf{sol}(\mathsf{EXISTS}(k_t^n = 1)) = \\ \{(\phi, \sigma) \mid \text{ there exists an } n \in \mathcal{N} \text{ and a } 0 \le t \le d \text{ such that } \phi(k_t^n) = 1\}. \end{array}$$

Finally, the constraint for the negation of the consequent C, written $\mathsf{NSAT}(C)$, is defined below. Here, the first three constraints match the three cases given above.

$$\begin{array}{lll} \mathsf{NSAT}(C) = & & \\ \&_{n \in \mathcal{N}}\ \&_{0 \le t \le d} & (\ \mathsf{IMPLIES}(\ \langle C \rangle(t)(n = 0) \text{ and } n_t = 0 \to k_t^n = 0\ )\ \ \& \\ & \ \ \mathsf{IMPLIES}(\ \langle C \rangle(t)(n = 1) \text{ and } n_t = 1 \to k_t^n = 0\ )\ \ \& \\ & \ \ \mathsf{IMPLIES}(\ \neg\langle C \rangle(t)(n = 0) \wedge \neg\langle C \rangle(t)(n = 1) \to k_t^n = 0\ )\ ) \\ \ \ \& & \mathsf{EXISTS}(k_t^n = 1) \end{array}$$

[1] As an optimization, in our implementation, variables are only introduced for those node and time point combinations that are actually referred to in the consequent.

Example 7. For $C = (a \to (\mathsf{reg}\ \mathbf{is}\ 0))$ **and** $(b \to (\mathsf{reg}\ \mathbf{is}\ 1))$, $\mathsf{NSAT}(C)$ is given by:

$$\begin{aligned} & \mathsf{IMPLIES}(\ a \text{ and } \mathsf{reg}_0 = 0 \to k_0^{\mathsf{reg}} = 0\ ) \\ \& \ & \mathsf{IMPLIES}(\ b \text{ and } \mathsf{reg}_0 = 1 \to k_0^{\mathsf{reg}} = 0\ ) \\ \& \ & \mathsf{IMPLIES}(\ \neg a \wedge \neg b \to k_0^{\mathsf{reg}}\ ) \ \ \& \ \ \mathsf{EXISTS}(k_t^n = 1) \end{aligned}$$

□

Proposition 3. *For every TEL-formula C:*

$$\mathsf{sol}(\mathsf{NSAT}(C)) = \{(\phi, \sigma \restriction \{0, 1, .., d\}) \mid \phi, \sigma \not\models C\}$$

The Constraint for an STE assertion is written $\mathsf{CEX}(c \models A \Longrightarrow C)$ and is defined by combining the trajectory constraint, the constraint for antecedent, and the constraint for the negation of the consequent.

$$\mathsf{CEX}(c \models A \Longrightarrow C) = \mathsf{TRAJ}(F_c, d) \ \ \& \ \ \mathsf{SAT}(A) \ \ \& \ \ \mathsf{NSAT}(C)$$

The correctness of the constraint formulation follows from Propositions 1,2 and 3.

Proposition 4. *For each circuit c and STE-assertion $A \Longrightarrow C$:*

$$c \models_{\to} A \Longrightarrow C \quad \Leftrightarrow \quad \mathsf{sol}(\mathsf{CEX}(c \models A \Longrightarrow C)) = \emptyset$$

5 Reducing constraints to SAT-problems

In this section, we show how we can instantiate the abstract constraints of the previous section to concrete SAT problems using a dual-rail encoding. First, we briefly restate the concept of a SAT-problem.

SAT problems A SAT-problem consists of set of *variables* W and a set of *clauses*. A *literal* is either a variable v or a negated variable $\bar{v}$. An *assignment* is a mapping $a : W \to \{0, 1\}$. For a negated variable $\bar{v}$, we define $a(\bar{v}) = \neg a(v)$. A clause, written $c = v_1 \vee v_2 \vee ... \vee v_n$, is said to be *satisfied* by an assignment a, if there exists an i such that $1 \leq i \leq n$ and $a(v_i) = 1$. A SAT-problem S is satisfied by an assignment a, written $a \models S$, if a satisfies every clause of S. The set of all satisfying assignments of a SAT-problem S is denoted $\mathsf{sa}(S)$.

SAT problem for an STE assertion Given an STE assertion $A \Longrightarrow C$ for a circuit c the SAT problem for the assertion is denoted $\mathsf{CEX}_{\mathsf{SAT}}(c \models A \Longrightarrow C)$. This *concrete* SAT-problem is build up from *concrete* primitive constraints in the same way as the *abstract* constraint $\mathsf{CEX}(c \models A \Longrightarrow C)$ is built up from primitive *abstract* constraints in the previous section. So, in this section we only need to show how the primitive abstract constraints can be instantiated to concrete SAT problems.

The SAT-problem generated for an STE-assertion of depth d contains a SAT-variable v for each variable v in the set of symbolic variables V. Furthermore, for each node n in the set of nodes $\mathcal{N}$ of the circuit c, and for each time point $0 \leq t \leq d$ *two* SAT-variables are introduced, written n_t^0 and n_t^1.

The two variables n_t^0 and n_t^1 encode the ternary value of node n at time t using a dual-rail encoding. If both variables are false, the value of node n_t is X. If n_t^0 is true, and n_t^1 is false, the node has value 0, if n_t^0 is false, and n_t^1 is true, the node has value 1. We exclude the possibility that both n^0 and n^1 are true by adding a clause $\overline{n_t^0} \vee \overline{n_t^1}$ to the SAT-problem for each n and t. The function mapping a dual-rail encoded ternary value to the ternary value itself, written tern, is defined by: $\mathsf{tern}(0,0) = \mathsf{X}$, $\mathsf{tern}(1,0) = 0$, and $\mathsf{tern}(0,1) = 1$.

Solutions A satisfying assignment a of such a SAT-problem is mapped to a solution (a tuple of an assignment of the symbolic variables and a sequence) by mapping the satisfying assignment a to the assignment of symbolic variables ϕ_a defined by $\phi_a(v) = a(v)$ and to a sequence σ_a defined by: $\sigma_a(t)(n) = \mathsf{tern}(a(n_t^0), a(n_t^1))$. So, the set of solutions for a SAT-problem is defined by: $\mathsf{sol}(S) = \{(\phi_a, \sigma_a) \mid a \in \mathsf{sa}(S)\}$

Concrete SAT-problems for the gates The SAT-problem for the AND-gate with inputs p_t and q_t and output r_t should have as solutions the sequences in which all forwards propagation has taken place. That is: (1) if $p_t = q_t = 1$ then $r_t = 1$, (2) if $p_t = 0$ then $r_t = 0$, and (3) if $q_t = 0$ then $r_t = 0$.

Recall that for each node n and time point t the clause $\overline{n_t^0} \vee \overline{n_t^1}$ is in the SAT-problem. This clause excludes the possibility that both n_t^0 and n_t^1 are true at the same time. Because of this, there first requirement can be captured in clauses by:

$$\overline{p_t^1} \vee \overline{q^1} \vee r_t^1$$

Now, the SAT-problem for the AND-gate, written $\mathsf{AND}_{\mathsf{SAT}}(p_t, q_t, r_t)$ is defined below. The problem consists of three clauses, corresponding to the three requirements above.

$$\mathsf{AND}_{\mathsf{SAT}}(p_t, q_t, r_t) = \{\overline{p_t^1} \vee \overline{q_t^1} \vee r_t^1, \overline{p_t^0} \vee r_t^0, \overline{q_t^0} \vee r_t^0\}$$

Note that these clauses do not yield backwards propagation of information. The assignment $r_t^1 = 1, r_t^0 = 0$ and $p_t^0 = p_t^1 = q_t^0 = q_t^1 = 0$ is a satisfying assignment of the clause set. So, the sequence that gives value 1 to the output of an AND-gate, but value X to its two inputs is a solution of the SAT-problem.

The following property states that the concrete SAT-problem for the AND-gate has the same solutions as the corresponding abstract constraint.

Proposition 5. *For all nodes p, q, and r, and time-point t:*

$$\mathsf{sol}(\mathsf{AND}_{\mathsf{SAT}}(p_t, q_t, r_t)) = \{(\phi, \sigma) | \ \sigma(t) \ \in \ F_{\mathrm{AND}}(p, q, r) \ \} = \mathsf{sol}(\mathsf{AND}(p_t, q_t, r_t).)$$

Concrete SAT-problems for comparing node values The SAT-problem for the constraint $\mathsf{LT}(n_{t_1} \leq m_{t_2})$ is defined below. The first clause makes sure that if node n has value 0 at time t, node m at time t_2 has that value as well. The next clause states the same requirement for value 1.

$$\mathsf{LT}_{\mathsf{SAT}}(n_{t_1} \leq m_{t_2}) = \{ \ \overline{n_{t_1}^0} \vee m_{t_2}^0, \quad \overline{n_{t_1}^1} \vee m_{t_2}^1 \ \}$$

Proposition 6. *For all $t_1, t_2 \in \mathbb{N}$ and $n, m \in \mathcal{N}$:*

$$\mathsf{sol}(\mathsf{LT}_{\mathsf{SAT}}(n_{t_1} \leq m_{t_2})) = \{ \ (\phi, \sigma) \mid \sigma(t_1)(n) \ \leq \ \sigma(t_2)(m) \ \} = \mathsf{sol}(\mathsf{LT}(n_{t_1} \leq m_{t_2})).$$

Concrete SAT-problems for Implications Methods to convert an arbitrary Boolean propositional formula to clauses are well-known. Typically, these methods introduce a fresh SAT-variable for each subexpression of the formula. Here, we abstract away from the details of such a method, and assume the existence of functions, cnf and lit that convert a Boolean propositional formula P on a set the set of variables V to a set of clauses $\mathsf{cnf}(P)$ on the set of variables $V' \supseteq V$ and a corresponding literal $\mathsf{lit}(p)$ such that (1) for all assignments $a : V \to \{0,1\}$ there exists an assignment $a' : V' \to \{0,1\}$ extending a such that $a' \models \mathsf{cnf}(P)$, and (2) for all assignments $a : V' \to \{0,1\}$ holds: $a \models \mathsf{cnf}(P) \Leftrightarrow a(\mathsf{lit}(P)) = a(P)$. Here $a(P)$ stands for the valuation of the expression P w.r.t. the assignment a.

Using these functions, the concrete SAT-problems for the implication constraints are defined. Given a Boolean propositional expression P, node $n \in \mathcal{N}$, time point $t \in \mathbb{N}$, the SAT problems for implications are defined as:

$$\begin{aligned}
&\mathsf{IMPLIES}_{\mathsf{SAT}}(\ P \to n_t = 0\ ) &&= \mathsf{cnf}(P) \cup \{\overline{\mathsf{lit}(P)} \vee n_t^0\}\\
&\mathsf{IMPLIES}_{\mathsf{SAT}}(\ P \to n_t = 1\ ) &&= \mathsf{cnf}(P) \cup \{\overline{\mathsf{lit}(P)} \vee n_t^1\}\\
&\mathsf{IMPLIES}_{\mathsf{SAT}}(\ P \to k_t^n = 0\ ) &&= \mathsf{cnf}(P) \cup \{\overline{\mathsf{lit}(P)} \vee \overline{k_t^n}\}\\
&\mathsf{IMPLIES}_{\mathsf{SAT}}(\ (P \text{ and } (n_t = 0)) \to k_t^n = 0\ ) &&= \mathsf{cnf}(P) \cup \{\overline{\mathsf{lit}(P)} \vee \overline{n_t^0} \vee \overline{k_t^n}\}\\
&\mathsf{IMPLIES}_{\mathsf{SAT}}(\ (P \text{ and } (n_t = 1)) \to k_t^n = 0\ ) &&= \mathsf{cnf}(P) \cup \{\overline{\mathsf{lit}(P)} \vee \overline{n_t^1} \vee \overline{k_t^n}\}
\end{aligned}$$

Proposition 7. *For each Boolean propositional expression P, node $n \in \mathcal{N}$, time point $t \in \mathbb{N}$ and $b \in \{0,1\}$, the following holds:*

$$\begin{aligned}
&\mathsf{sol}(\mathsf{IMPLIES}_{\mathsf{SAT}}(\ P \to n_t = b\ )) &&= \mathsf{sol}(\mathsf{IMPLIES}(\ P \to n_t = b\ ))\\
&\mathsf{sol}(\mathsf{IMPLIES}_{\mathsf{SAT}}(\ P \to k_t^n = 0\ )) &&= \mathsf{sol}(\mathsf{IMPLIES}(\ P \to k_t^n = 0\ ))\\
&\mathsf{sol}(\mathsf{IMPLIES}_{\mathsf{SAT}}(\ (P \text{ and } n_t = b) \to k_t^n = 0\ )) = \\
&\qquad\qquad \mathsf{sol}(\mathsf{IMPLIES}(\ (P \text{ and } n_t = b) \to k_t^n = 0\ ))
\end{aligned}$$

Finally, the concrete SAT-problem for the abstract constraint $\mathsf{EXISTS}(k_t^n = 1)$ is needed. The constraint is constructed as a disjunction of all k_t^n where n ranges over the set of nodes of the circuit, and t over the time points 0 to d.

$$\mathsf{EXISTS}_{\mathsf{SAT}}(k_t^n = 1) = \vee_{n \in \mathcal{N}} . \vee_{0 \le t \le d}\ k_t^n$$

Proposition 8. $\mathsf{sol}(\mathsf{EXISTS}_{\mathsf{SAT}}(k_t^n = 1)) = \mathsf{sol}(\mathsf{EXISTS}(k_t^n = 1))$

6 Constraint vs Simulation based SAT-STE

The main difference between simulation-based SAT-STE and constraint-based SAT-STE is that the first generates a SAT problem representing the set of *weakest* trajectories satisfying the antecedent but not the consequent, while the latter generates a SAT-problem that represents *all* such trajectories. For this reason, simulation based SAT-STE generates much larger SAT-problems.

The difference in generated SAT-problems can be illustrated by considering a single AND-gate with input nodes p and q and output r. This AND-gate is assumed to be part of a larger circuit, but here we consider only the clauses generated for the AND-gate.

		Verification Time(s)			**Bug Finding Time(s)**			**#variables** $(\times 10^3)$			**#clauses** $(\times 10^3)$		
	#nodes $(\times 10^3)$	BMC	CON	SIM	BMC	CON	SIM	BMC	CON	SIM	BMC	CON	SIM
shifter-64	5	8.4	**2.4**	17	0.0	0.0	0.0	5	9	9	19	24	41
shifter-128	19	175	**36**	364	**0.0**	**0.0**	0.1	18	35	35	72	89	158
shifter-256	71	3443	**500**	8127	1.6	**0.3**	1.1	69	137	137	275	343	613
shifter-512	275	time out	**5621**	time out	3.8	**0.7**	1.6	271	537	537	1101	1344	2451
mem-10-4	27	13	**12**	21	9.4	**8.1**	17	18	27	27	47	51	84
mem-11-4	55	78	**47**	83	**24**	44	82	37	53	53	94	102	168
mem-12-4	115	367	**222**	435	371	**157**	197	74	107	107	188	205	336
mem-13-4	238	2215	**876**	1449	1947	**564**	1087	147	213	213	377	410	672
mem-14-4	492	8066	**3612**	5524	9626	**1970**	3194	295	426	426	754	819	1343
treemem-10-4	14	2.6	**0.6**	3.7	**0.0**	0.4	3.7	14	18	18	39	39	63
treemem-11-4	29	5.0	**3.9**	15	**0.1**	4.4	7.3	29	37	37	78	78	127
treemem-12-4	57	22	**21**	62	22	**21**	17	57	74	74	156	156	254
treemem-13-4	115	106	**107**	281	**98**	102	160	115	147	147	311	311	508
treemem-14-4	229	476	**452**	1153	434	**427**	1059	229	295	295	623	623	1016
con-6-10-4	15	**0.9**	**0.9**	4.3	0.7	**0.6**	1.1	15	20	20	41	41	67
con-6-11-4	30	**3.9**	5.1	16	**1.7**	2.0	13	30	39	39	82	82	135
con-6-12-4	61	**22**	25	70	**12**	17	40	60	78	78	164	165	270
con-7-13-4	118	**116**	123	298	97	**49**	70	118	153	153	320	321	525
con-7-14-4	237	**431**	512	1170	**204**	257	665	236	305	305	641	643	1051

Fig. 4. Benchmarks on instances of generically-sized circuits.

In constraint based SAT-STE, clauses are generated that make sure that the solutions represent *all* trajectories. In simulation-based SAT-STE however, the set of solutions to the SAT-problem represents only the set of *weakest* trajectories. Therefore, the clauses for the AND-gate do not only contain the clauses mentioned in Sect. 5, but also require the following: if forward propagation cannot derive a Boolean value for the output, then the output has value X. The following extra requirements are thus generated: if $p = q = \mathsf{X}$ then $r = \mathsf{X}$, if $p = \mathsf{X}$ and $q = 1$ then $r = \mathsf{X}$, and if $p = 1$ and $q = \mathsf{X}$ then $r = \mathsf{X}$. So, for an AND-gate, simulation-based SAT-STE requires twice as many clauses as constraint-based STE. A similar result holds for other gates. Therefore, simulation-based SAT-STE generates much larger SAT-problems than constraint-based STE.

Optimization An advantage of STE is that when model checking a small part of a large circuit (for instance an adder within a complete microprocessor) we can set the inputs to the irrelevant parts of the circuit to X. Then, during simulation, all node values of the irrelevant parts receive value X, and only the values of the nodes in the part of interest are represented in the resulting symbolic expressions for the weakest trajectory.

In our algorithm, we represent all trajectories. Therefore, in the pure form of the algorithm, constraints are generated for all gates, even for the gates for which the output node would directly receive value X in a simulation based algorithm. Therefore, we apply a simple and light-weight optimization to our algorithm: if symbolic simulation yields a scalar value ($0, 1$ or X) for a node, the node receives this value in our algorithm and no constraints are generated for the gates driving the node. For all other gates constraints are generated as described in Sect. 5.

		Verification Time(s)				**#variables** $(\times 10^3)$			**#clauses** $(\times 10^3)$		
	#nodes $(\times 10^3)$	BDD	BMC	CON	SIM	BMC	CON	SIM	BMC	CON	SIM
cam (full enc.)	5	time out	1.6	1.6	1.6	4	4	4	8	8	8
cam (plain enc.)	5	time out	1.8	0.9	11	4	5	6	10	11	16
cam (cam enc.)	5	0.1	2.6	2.4	4.2	4	5	6	8	10	16
mem	25	0.3	11	13	23	41	43	60	109	101	175

Fig. 5. Benchmarks on circuits from Intel's GSTE tutorial.

7 Results

We have implemented two algorithms: CON-SAT STE, performing constraint-based SAT-STE, and SIM-SAT STE, performing simulation-based SAT-STE. We compare the CON-SAT algorithm and SIM-SAT algorithms with each other.

As a reference point, we also compare with Bounded Model Checking (BMC) [2]. BMC can be used to verify STE assertions by interpreting the assertion as an LTL formula; the completeness threshold [2] is simply the depth of the assertion. Note that BMC solves a different problem, as it does not use STE's three-valued abstraction.

To make the comparison between the algorithms fair, the same SAT-solver (the latest version of MiniSAT [8]) is used for all methods. The benchmarks were run on a cluster of PCs with AMD Barton XP2800+ processors and each with one gigabyte of memory.

First, we performed benchmarks on instances of generically-sized circuits, designed by ourselves. The properties we consider for these circuits are: (1) shifter-w; for a variable shifter of width w, full correctness using *symbolic indexing* [5], (2) (tree-)mem-a-d; for a (tree shaped) memory with address width a and data width w, the property that reading an address after writing a value to it yields the same value, and (3) con-c-a-d; for a memory controller with a cache of address width c, a memory of address width a and data width d, the property that reading an address after writing yields the same value, both for the cache and the memory. The times needed to solve the problems and the numbers of variables and clauses in each SAT-problem are given in Fig. 4.

The results show, as expected, that the number of SAT variables for CON-SAT-STE and SIM-SAT-STE are about equal — two variables are introduced for each relevant node and time point. Also as expected, the number of clauses is much larger for SIM-SAT-STE, as explained in Sect. 6. Furthermore, CON-SAT-STE solves the the STE problems much faster than SIM-SAT-STE, something we believe is caused by the reduction in problem size.

For the shifter-n and mem-a-d benchmarks, CON-SAT STE performs better than BMC. For the tree-mem-a-d and con-c-a-d benchmarks the two methods perform comparably. So, in some cases the abstractions used in STE can be beneficial when using SAT-based methods. The reader should, however, realize that the point of this paper is not to advocate the usage of SAT-based STE over BMC or BDD-based STE. Bjesse et al. and Singerman et al. have already shown that SAT-based STE is a useful complement to BDD-based STE and BMC in industrial settings [3, 9]. The point of this paper is to present an algorithm that improves upon the algorithms used by Bjesse and Singerman.

The second set of circuits have been supplied to us by Intel Strategic CAD Labs. The circuits are part of a tutorial for GSTE. In Fig. 5 we compare the performance of BMC, BDD-based STE, SIM-SAT-STE and CON-SAT-STE for the verification of several properties of the Content Addressable Memory (CAM) and the memory circuit from this tutorial. Forte [4] was used to perform BDD-based STE. For the CAM, we verify the associative read property using three symbolic indexing schemes from Pandey et al [5]. The CAM contains 16 entries, has a data-width of 64 bits and a tag-width of 8 bits. For the memory, the property that reading address D after writing value V to address D yields value V is verified. Standard symbolic indexing is used. The memory has an address-width of 6 bits, and a data-width of 128 bits.

Pandey et al. show in [5] that verifying the associative read property of CAMs using BDD-based STE is highly non-trivial. The problem is that the straight-forward specification (which they call the *full encoding*) of the property leads to a BDD blowup. They present an improved specification, called the *plain encoding*, that results in smaller BDDs, but that still causes a BDD blow up. Only the most efficient (and complex) specification they introduce, called the *cam encoding*, yields small enough BDDs to make verification of the property go through.

Also for these benchmarks, CON-SAT-STE produces smaller and easier to solve SAT-problems then SIM-SAT-STE. Moreover, the experiments confirm the results of Pandey et al: BDD-based STE cannot be used to verify CAMs using the full or plain encoding. In these experiments, the performance of SAT-based STE is more robust. No matter which encoding is used for verifying the associative read property of the CAM, the SAT-based methods manage to verify the property. This can be explained as follows. The efficiency of a BDD-based STE verification run is highly dependent on the number of variables in the BDDs involved. BDD-based verification methods are usually not able to handle problems with more than several hundred variables. Therefore, symbolic indexing methods minimizing the number of symbolic variables in an STE-assertion are crucial to the efficiency of BDD-based STE. SAT-solvers, on the other hand, have proved to be much less dependent on the number of variables. Therefore, symbolic indexing techniques, minimizing the number of variables, are much less relevant for SAT-based STE.

Reflection Constraint-based SAT-STE generates smaller problems that are easier to solve than simulation-based SAT-STE, on all our benchmarks. We realize that the problem set we used is quite limited, but we believe it nevertheless indicates the usefulness of our approach.

Plain BMC sometimes outperforms SAT-based STE. Although this is an interesting observation, BMC cannot replace SAT-based STE because it implements a different semantics. For instance, at Intel, the satGSTE tool is used to help develop specifications in GSTE model checking [9]. Here, SAT-based STE is used to get quick feedback when debugging or refining a GSTE assertion graph. In this setting, it is *essential* to have a model checking method that implements the *same semantics* as BDD-based STE, but is not as sensitive to BDD-blow up . This is where SAT-based STE comes in.

8 Conclusions and Future Work

Bjesse et al. and Singerman et al. have shown that SAT-based STE is a useful complement to BDD-based STE and BMC in industrial settings [3, 9]. Their algorithms are based on simulation, and generate a SAT-problem that represents the set of weakest trajectories satisfying the antecedent but not the consequent of an STE assertion.

We have presented a new constraint-based SAT-algorithm for STE. Instead of generating a SAT-problem that represents the set of weakest trajectories satisfying the antecedent but not the consequent, our algorithm generates a SAT problem whose solutions represent *all* trajectories satisfying the antecedent but not the consequent. The advantage of representing the set of all such trajectories in the SAT problem (instead of just the weakest trajectories) is that smaller SAT-problems are generated.

Benchmarks, both on circuits designed by ourselves and on circuits taken from Intel's GSTE tutorial, show that our constraint based SAT algorithm for STE performs significantly better than current simulation based algorithms.

Future work Intel's satGSTE tool [9] is a bug finding method for GSTE, it implements a *bounded* version of GSTE: only a finite subset of all finite paths in a GSTE assertion graph is considered. Currently the core of the satGSTE tool is a simulation-based SAT-STE algorithm. We conjecture that replacing the tool with a constraint-based SAT-STE algorithm might significantly improve the performance of the tool.

Furthermore, we would like to investigate whether we can use SAT for doing full (unbounded) GSTE model checking. Finally, in (G)STE finding the right specification can be very time consuming. Therefore, we would like to investigate whether SAT can be used to implement a form of *automatic specification refinement* for (G)STE.

Acknowledgment We are grateful for an equipment grant from Intel Corporation.

References

1. Mark Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, 2000.
2. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.
3. P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference of Computer-Aided Verification*, 2001.
4. http://www.intel.com/software/products/opensource/tools1/verification.
5. Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *34th Design Automation Conference (DAC'97)*, pages 167–172. Association for Computing Machinery.
6. Tom Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of the 40th conference on Design automation*, pages 1–6. ACM Press, 2003.
7. Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design: An International Journal*, 6(2):147–189, March 1995.
8. Niklas Eén & Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, 2003.
9. Jin Yang, Rami Gil, and Eli Singerman. satGSTE: Combining the abstraction of GSTE with the capacity of a SAT solver. In *Designing Correct Circuits (DCC'04)*, 2004.
10. Jin Yang and C.-J. H. Seger. Introduction to generalized symbolic trajectory evaluation. In *IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD '01)*, pages 360–367, Washington - Brussels - Tokyo, September 2001. IEEE.