Automatic Refinement and Vacuity Detection for Symbolic Trajectory Evaluation

Rachel Tzoref[1,2] and Orna Grumberg[1]
{rachelt,orna}@cs.technion.ac.il

[1] Computer Science Department, Technion, Haifa, Israel
[2] IBM Haifa Research Laboratory, Israel

Abstract. Symbolic Trajectory Evaluation (STE) is a powerful technique for model checking. It is based on 3-valued symbolic simulation, using 0,1 and X ("unknown"). The X value is used to abstract away parts of the circuit. The abstraction is derived from the user's specification. Currently the process of abstraction and refinement in STE is performed manually. This paper presents an automatic refinement technique for STE. The technique is based on a clever selection of constraints that are added to the specification so that on the one hand the semantics of the original specification is preserved, and on the other hand, the part of the state space in which the "unknown" result is received is significantly decreased or totally eliminated. In addition, this paper raises the problem of vacuity of passed and failed specifications. This problem was never discussed in the framework of STE. We describe when an STE specification may vacuously pass or fail, and propose a method for vacuity detection in STE.

1 Introduction

Symbolic Trajectory Evaluation (STE) [11] is a powerful technique for hardware model checking. STE is based on combining 3-valued simulation with symbolic simulation. It is applied to a circuit M, described as a graph over *nodes* (gates and latches). The specification consists of assertions in a restricted temporal language. The assertions are of the form $A \Longrightarrow C$, where the *antecedent* A expresses constraints on nodes n at different times t, and the *consequent* C expresses requirements that should hold on such nodes (n, t). STE computes a symbolic representation for each node (n, t). The size of this representation depends on the size of A, rather than on the circuit size. *Abstraction* in STE is derived from the specification by initializing all inputs not appearing in A to the X ("unknown") value. A forth value, $\perp$, represents a contradiction between the constraint of A on some node (n, t) and its actual behavior. A *refinement* amounts to changing the assertion in order to present nodes values more accurately.

STE assertions may either pass or fail on M. In [5], a 4-valued truth domain $\{0, 1, X, \perp\}$ is defined for the temporal language of STE, corresponding to the 4-valued domain of the values of the circuit nodes. The motivation for a 4-valued semantics is to distinguish between different causes for the pass or fail of an STE assertion. The X truth value distinguishes the case in which the STE assertion fails due to partial information about the state space from the case in which it is actually violated by M. In the latter case a *counterexample* is produced, representing an execution of M that satisfies A but contradicts C. The X truth value stems from a too coarse antecedent which underspecifies the circuit. The $\perp$ truth value indicates that the STE assertion passes vacuously due to a contradiction between A and M.

Generalized STE (GSTE) [19] is a significant extension of STE that can verify all ω-regular properties. Manual refinement methods for GSTE are presented in [18]. In [16], SAT-based STE is used for manual refinement of GSTE assertion graphs.

(G)STE has been in active use in the industry, and has been very successful in verifying huge circuits containing large data paths [12, 10, 17]. Its main drawback, however, is the need for manual abstraction and refinement, which can be very labor-intensive.

Our Contribution. We propose a technique for automatic refinement of assertions in STE. In our technique, the initial abstraction is derived, as usual in STE, from the given specification. The refinement is an iterative process, which stops when a truth value other than X is achieved. In case of a 0 truth value, a counterexample is presented to the user. Our automatic refinement is applied when the STE specification results with X. We compute a set of input nodes, whose refinement is sufficient for eliminating the X truth value. We further suggest heuristics for choosing a small subset of this set.

Selecting a ”right” set of inputs has a crucial role in the success of the abstraction and refinement process: selecting too many inputs will add many variables to the computation of the symbolic representation, and may result in memory and time explosion. On the other hand, selecting too few inputs or selecting inputs that do not affect the result of the verification will lead to many iterations with an X truth value.

We point out that, as in any automated verification framework, we are limited by the following observations. First, there is no automatic way to determine whether the provided specification is correct. Therefore, we assume it is, and we make sure that our refined assertion passes on the concrete circuit iff the original assertion does. Second, bugs cannot automatically be fixed. Thus, counterexamples are analyzed by the user.

Abstraction-Refinement is a well known methodology in model checking [4, 6] for fighting the state explosion problem. In [3], it is shown that the abstraction in STE is an abstract interpretation via a Galois connection. [9] presents a SAT-based algorithm to assist in manual refinement of STE assertions. However, automatic refinement has never been suggested before for STE. The work that is closest to ours is [15], which suggests an automatic abstraction-refinement for symbolic simulation. However, the suggested heuristics are significantly different from ours.

Another important contribution of our work is identifying that STE results may hide vacuity. This possibility was never raised before. Hidden vacuity may occur since an abstract execution of M on which the truth value of the specification is 1 or 0, might not correspond to any concrete execution of M. In such a case, a pass is *vacuous*, while a counterexample is *spurious*. We propose a method for detecting these cases.

We implemented our automatic refinement technique within Intel’s Forte environment [12]. We ran it on two nontrivial circuits with several assertions. Our experimental results show success in automatically identifying a set of inputs that are crucial for reaching a definite truth value. Thus, a small number of iterations were needed.

2 Basic Definitions

A circuit M consists of a set of nodes $\mathcal{N}$, connected by directed edges. The nodes consist of inputs and internal nodes. Internal nodes consist of latches and combinational nodes. Each combinational node is associated with a Boolean function. We say that a node n_1 enters a node n_2 if there exists a directed edge from n_1 to n_2. The nodes

entering a certain node are its ***source nodes***, and the nodes to which a node enters are its ***sink nodes***. The value of a latch at time t can be expressed as a Boolean expression over its source nodes at times t and $t-1$, and over the latch value at time $t-1$. The directed graph induced by M may contain loops but no combinational loops.
Throughout the paper we refer to a node n at a specific time t as (n,t).

The ***bounded cone of influence*** **(BCOI)** of a node (n,t) contains all nodes (n',t') with $t' \leq t$ that may influence the value of (n,t), and is defined recursively as follows: the BCOI of a combinational node at time t is the union of the BCOI of its source nodes at time t, and the BCOI of a latch at time t is the union of the BCOI of its source nodes at times t and $t-1$ according to the latch type.

Usually, the circuit nodes receive Boolean values. In STE, a third value, X ("unknown"), is introduced. Attaching X to a certain node represents lack of information regarding the truth value of that node. A forth value, $\bot$, is added to represent the over-constrained value, in which a node is forced both to 0 and to 1. This value indicates that contradiction exists between external assumptions on the circuit and its actual behavior. The set of values $\mathcal{Q} \equiv \{0,1,X,\bot\}$ forms a complete lattice with the partial order $0 \sqsubseteq X$, $1 \sqsubseteq X$, $\bot \sqsubseteq 0$ and $\bot \sqsubseteq 1$. This order corresponds to set inclusion, where X represents the set $\{0,1\}$, and $\bot$ represents the empty set. As a result, the *greatest lower bound* $\sqcap$ corresponds to set intersection and the *least upper bound* $\sqcup$ corresponds to set union. The Boolean operations AND, OR and NOT are extended to the domain $\mathcal{Q}$ as shown in Figure 1.

AND	X	0	1	$\bot$
X	X	0	X	$\bot$
0	0	0	0	$\bot$
1	X	0	1	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

OR	X	0	1	$\bot$
X	X	X	1	$\bot$
0	X	0	1	$\bot$
1	1	1	1	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

NOT	
X	X
0	1
1	0
$\bot$	$\bot$

Fig. 1. Quaternary operations

A ***state*** s of the circuit M is an assignment of values from $\mathcal{Q}$ to all circuit nodes, $s : \mathcal{N} \rightarrow \mathcal{Q}$. Given two states s_1, s_2, we say that $s_1 \sqsubseteq s_2 \iff ((\exists n \in \mathcal{N} : s_1(n) = \bot) \vee (\forall n \in \mathcal{N} : \; s_1(n) \sqsubseteq s_2(n)))$. A state is ***concrete*** if all nodes are assigned with values out of $\{0,1\}$. A state s is an abstraction of a concrete state s_c if $s_c \sqsubseteq s$.

A ***sequence*** σ is any infinite series of states. We denote by $\sigma(i), i \in \mathbb{N}$, the state at time i in σ, and by $\sigma(i)(n), i \in \mathbb{N}, n \in \mathcal{N}$, the value of node n in the state $\sigma(i)$. $\sigma^i, i \in \mathbb{N}$, denotes the suffix of σ starting at time i. We say that $\sigma_1 \sqsubseteq \sigma_2 \iff ((\exists i \geq 0, n \in \mathcal{N} : \sigma_1(i)(n) = \bot) \vee (\forall\, i \geq 0 : \sigma_1(i) \sqsubseteq \sigma_2(i)))$. Note that we refer to states and sequences that contain $\bot$ values as least elements w.r.t $\sqsubseteq$.

Let V be a set of symbolic Boolean variables over the domain $\{0,1\}$. A ***symbolic expression*** over V is an expression consisting of quaternary operations, applied to $V \cup \mathcal{Q}$. A ***symbolic state*** over V is a mapping which maps each node of M to a symbolic expression. Each symbolic state represents a set of states, one for each assignment to the variables in V. A ***symbolic sequence*** over V is a series of symbolic states. It represents a set of sequences, one for each assignment to V. Given a symbolic sequence σ and an assignment ϕ to V, $\phi(\sigma)$ denotes the sequence that is received by applying ϕ to all symbolic expressions in σ. Given two symbolic sequences σ_1,σ_2 over V, we say that $\sigma_1 \sqsubseteq \sigma_2$ if for all assignments ϕ to V, $\phi(\sigma_1) \sqsubseteq \phi(\sigma_2)$.
A ***Trajectory Evaluation Logic*** (TEL) formula is defined recursively over V as follows:

$$f ::= n \text{ is } p \mid f_1 \wedge f_2 \mid p \rightarrow f \mid \mathbf{N} f$$

where $n \in \mathcal{N}$, p is a Boolean expression over V and $\mathbf{N}$ is the next time operator. Note that TEL formulas can be expressed as a finite set of constraints on values of specific nodes at specific times. N^t denotes the application of t next time operators. The constraints on (n, t) are those appearing in the scope of N^t. The ***maximal depth*** of a TEL formula f, denoted depth(f), is the maximal time t for which a constraint exists in f on some node (n, t), plus 1.

Usually, the satisfaction of a TEL formula f on a symbolic sequence σ is defined in the 2-valued truth domain [11], i.e., f is either satisfied or not satisfied. In [5], $\mathcal{Q}$ is used also as a 4-valued truth domain for an extension of TEL. Our 4-valued semantics definition is different from [5] w.r.t $\perp$ values. In [5], a sequence σ containing $\perp$ values could satisfy f with a truth value different from $\perp$. In our definition this is not allowed. We believe that our definition captures better the intent behind the specification w.r.t contradictory information about the state space. Given a TEL formula f over V, a symbolic sequence σ over V, and an assignment ϕ to V, we define the satisfaction of f as follows:
$[\phi, \sigma \models f] = \perp \;\leftrightarrow\; \exists i \geq 0, n \in \mathcal{N} : \phi(\sigma)(i)(n) = \perp$. Otherwise:
$[\phi, \sigma \models n \text{ is } p] = 1 \;\leftrightarrow \phi(\sigma)(0)(n) = \phi(p)$
$[\phi, \sigma \models n \text{ is } p] = 0 \;\leftrightarrow\; \phi(\sigma)(0)(n) \neq \phi(p)$ and $\phi(\sigma)(0)(n) \in \{0, 1\}$
$[\phi, \sigma \models n \text{ is } p] = X \;\leftrightarrow\; \phi(\sigma)(0)(n) = X$ $\qquad \phi, \sigma \models p \rightarrow f = (\neg\phi(p) \vee \phi, \sigma \models f)$
$\phi, \sigma \models f_1 \wedge f_2 = (\phi, \sigma \models f_1 \wedge \phi, \sigma \models f_2)$ $\qquad \phi, \sigma \models \mathbf{N}f = \phi, \sigma^1 \models f$
Note that given an assignment ϕ to V, $\phi(p)$ is a constant (0 or 1). In addition, the $\perp$ truth value is determined only according to ϕ and σ, regardless of f. It is proven in [5] that the satisfaction relation is monotonic, i.e., for all TEL formulas f, symbolic sequences σ_1, σ_2 and assignments ϕ to V, if $\phi(\sigma_2) \sqsubseteq \phi(\sigma_1)$ then $[\phi, \sigma_2 \models f] \sqsubseteq [\phi, \sigma_1 \models f]$. This also holds for our satisfaction definition. We define the truth value of $\sigma \models f$ as follows:
$[\sigma \models f] = 0 \;\leftrightarrow\; \exists\phi : [\phi, \sigma \models f] = 0$
$[\sigma \models f] = X \;\leftrightarrow\; \forall\phi : [\phi, \sigma \models f] \neq 0$ and $\exists\phi : [\phi, \sigma \models f] = X$
$[\sigma \models f] = 1 \;\leftrightarrow\; \forall\phi : [\phi, \sigma \models f] \notin \{0, X\}$ and $\exists\phi : [\phi, \sigma \models f] = 1$
$[\sigma \models f] = \perp \;\leftrightarrow\; \forall\phi : [\phi, \sigma \models f] = \perp$

It is proven in [5] that every TEL formula f has a ***defining sequence***, which is a symbolic sequence σ^f so that $[\sigma^f \models f] = 1$ and for all σ, $[\sigma \models f] \in \{1, \perp\}$ iff $\sigma \sqsubseteq \sigma^f$. For example, $\sigma^{q \rightarrow (n \text{ is } p)}$ is the sequence $s_{(n,q\rightarrow p)} s_x s_x s_x ...$, where $s_{(n,q\rightarrow p)}$ is the state in which n equals $(q \rightarrow p) \wedge (\neg q \rightarrow X)$, and all other nodes equal X, and s_x is the state in which all nodes equal X. σ^f may be incompatible with the behavior of M. A ***(symbolic) trajectory*** π is a (symbolic) sequence that is compatible with the behavior of M [8]: let $val(n, t)$ be the value of a node (n, t) as computed according to its source nodes values in π. It is required that for all nodes (n, t), $\pi(t)(n) \sqsubseteq val(n, t)$ (strict equality is not required in order to allow external assumptions on nodes values to be embedded into π). A trajectory is ***concrete*** if all its states are concrete. A trajectory π is an abstraction of a concrete trajectory π_c if $\pi_c \sqsubseteq \pi$.

The ***defining trajectory*** π^f of M and f is a symbolic trajectory so that $[\pi^f \models f] \in \{1, \perp\}$ and for all trajectories π of M, $[\pi \models f] \in \{1, \perp\}$ iff $\pi \sqsubseteq \pi^f$ (Similar definitions for σ^f and π^f exist in [11] w.r.t a 2-valued truth domain). Given σ^f, π^f is computed as follows: $\forall i$, $\pi^f(i)$ is initialized to $\sigma^f(i)$, and the nodes values from time i and $i - 1$ are propagated forward to nodes at time i until no new values are derived. The $\sqcap$ operator is used to incorporate a propagated value into the current value of a node (n, i).

STE assertions are of the form $A \Longrightarrow C$, where A (the antecedent) and C (the consequent) are TEL formulas. A expresses constraints on circuit nodes at specific times, and C expresses requirements that should hold on circuit nodes at specific times. $M \models (A \Longrightarrow C)$ iff for all concrete trajectories π of M and assignments ϕ to V, $[\phi, \pi \models A] = 1$ implies that $[\phi, \pi \models C] = 1$.

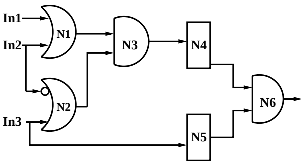

Fig. 2. A Circuit

A natural verification algorithm for an STE assertion $A \Longrightarrow C$ is to compute the defining trajectory π^A of M and A and then compute the truth value of $\pi^A \models C$. If $[\pi^A \models C] \in \{1, \perp\}$ then it holds that $M \models (A \Longrightarrow C)$. If $[\pi^A \models C] = 0$ then it holds that $M \not\models (A \Longrightarrow C)$. If $[\pi^A \models C] = X$, then it cannot be determined whether $M \models (A \Longrightarrow C)$. The case in which there is ϕ so that $\phi(\pi^A)$ contains $\perp$ is known as an ***antecedent failure***. The default behavior of most STE implementations is to consider antecedent failures as illegal, and the user is required to change A in order to eliminate any $\perp$ values. For lack of space, in the rest of the paper, we take the same approach. The alternative approach of STE implementations that supports occurrences of $\perp$ in π^A is described in [13]. Note that although π^A is infinite, it is suffice to examine only a bounded prefix of length depth(A) in order to detect $\perp$ in π^A. The first $\perp$ in π^A is the result of the $\sqcap$ operation on some node (n, t), where both operands have contradicting assignments 0 and 1. Since $\forall i >$ depth(A) : $\sigma^A(i) = s_x$, it must hold that $t \leq$ depth(A). In order to compute $\pi^A \models C$ (assuming π^A does not contain $\perp$), π^A is compared to σ^C, the defining sequence of C. If $\pi^A \sqsubseteq \sigma^C$, then $[\pi^A \models C] = 1$. If there are $\phi, i \geq 0, n \in \mathcal{N}$ so that $\phi(\pi^A)(i)(n) \not\sqsubseteq \phi(\sigma^C)(i)(n)$ and $\phi(\pi^A)(i)(n) \not\sqsupseteq \phi(\sigma^C)(i)(n)$, then $[\pi^A \models C] = 0$. Otherwise, $[\pi^A \models C] = X$. Note that although π^A and σ^C are infinite, it is suffice to examine only a bounded prefix of length depth(C), since $\forall i >$ depth(C) : $\sigma^C(i) = s_x$.

Example 1. Consider the circuit M in Figure 2, containing three inputs In1, In2 and In3, two OR nodes N1 and N2, two AND nodes N3 and N6, and two latches N4 and N5. For simplicity, the latches clocks were omitted and at each time t the latches sample their data source node from time $t-1$. Note the negation on the source node In2 of N2. Also consider the STE assertion $A \Longrightarrow C$, where $A =$ (In1 is 0)$\wedge$(In3 is v_1)$\wedge$(N3 is 1), and $C = \mathbf{N}$(N6 is 1). Figure 3 describes the defining trajectory π^A of M and A, up to time 1. It contains the symbolic expression of each node at time 0 and 1. The state $\pi^A(i)$ is represented by row i. The notation $v_1?1 : X$ stands for "if v_1 holds then 1 else X". σ^C is the sequence in which all nodes at all times are assigned X, except for node N6 at time 1, which is assigned 1. $[\pi^A \models C] = 0$ due to those assignments in which $v_1 = 0$. We will return to this example in Section 5.

Time	In1	In2	In3	N1	N2	N3	N4	N5	N6
0	0	X	v_1	X	$v_1?1 : X$	1	X	X	X
1	X	X	X	X	X	X	1	v_1	v_1

Fig. 3. The Defining Trajectory π^A

STE implementations use a specific encoding called ***dual rail*** in order to represent the nodes (n, t) in sequences. The dual rail of a node (n, t) in π^A consists of two functions defined from V to $\{0, 1\}$: $f^1_{n,t}$ and $f^0_{n,t}$, where V is the set of variables appearing in A. For each assignment ϕ to V, if $f^1_{n,t} \wedge \neg f^0_{n,t}$ holds under ϕ, then (n, t) equals 1 under ϕ. Similarly, $\neg f^1_{n,t} \wedge f^0_{n,t}$, $\neg f^1_{n,t} \wedge \neg f^0_{n,t}$ and $f^1_{n,t} \wedge f^0_{n,t}$ stand for 0, X and $\perp$ under ϕ, respectively. Likewise, $g^1_{n,t}$ and $g^0_{n,t}$ is the dual rail representation of (n, t) in σ^C. Note that $g^1_{n,t} \wedge g^0_{n,t}$ never holds, since we always assume that C is not self-contradicting.

3 Choosing Our Automatic Refinement Methodology

Intuitively, the defining trajectory π^A of a circuit M and an antecedent A is an abstraction of all concrete trajectories of M on which the consequent C is required to hold. This abstraction is directly derived from A. If $[\pi^A \models C] = X$, then A is too coarse, that is, contains too few constraints on the values of circuit nodes. Our goal is to automatically refine A (and subsequently π^A) in order to eliminate the X truth value.

In this section we examine the requirements that should be imposed on automatic refinement in STE. We then describe our automatic refinement methodology, and formally state the relationship between the two abstractions, derived from the original and refined antecedent. We refer only to STE implementations that compute π^A. We assume that antecedent failures are handled as described in Chapter 2.

Traditionally, the abstraction and refinement process in STE works as follows: the user writes an STE assertion $A \Longrightarrow C$ for M, and receives a result from STE. If $[\pi^A \models C] = 0$, then the set of all ϕ so that $[\phi, \pi^A \models C] = 0$ is provided to the user. This set, called the ***symbolic counterexample***, is given by the Boolean expression over V: $\bigvee_{(n,t)\in C}((g^1_{n,t} \wedge \neg f^1_{n,t} \wedge f^0_{n,t}) \vee (g^0_{n,t} \wedge f^1_{n,t} \wedge \neg f^0_{n,t}))$. It stems from either an illegal behavior of the circuit, or an erroneous specification. The user decides which of these possibilities the counterexample displays. If $[\pi^A \models C] = X$, then the set of all ϕ so that $[\phi, \pi^A \models C] = X$ is provided to the user. This set, called the ***symbolic incomplete trace***, is given by: $\bigvee_{(n,t)\in C}((g^1_{n,t} \vee g^0_{n,t}) \wedge \neg f^1_{n,t} \wedge \neg f^0_{n,t})$. The user decides how to refine the specification in order to eliminate the partial information that causes the X truth value. Otherwise, $[\pi^A \models C] = 1$ and the verification completes successfully.

As mentioned before, we must assume that the given specification is correct. Thus, automatic refinement of A must preserve the semantics of $A \Longrightarrow C$: Let $A_{new} \Longrightarrow C$ denote the refined assertion. Let $runs(M)$ denote the set of all concrete trajectories of M. We require that $A_{new} \Longrightarrow C$ holds on $runs(M)$ iff $A \Longrightarrow C$ holds on $runs(M)$.

In order to achieve the above preservation, we chose our automatic refinement as follows. Whenever $[\pi^A \models C] = X$, we add constraints to A that force the value of input nodes at certain times (and initial values of latches) to the value of *fresh symbolic variables*, that is, symbolic variables that do not already appear in V. By initializing an input (in, t) with a fresh symbolic variable instead of X, we represent the value of (in, t) accurately and add knowledge about its effect on M. However, we do not constrain input behavior that was allowed by A, nor do we allow input behavior that was forbidden by A. Thus, the semantics of A is preserved. In Section 4, a small but significant addition is made to our refinement technique.

We now formally state the relationship between the abstractions derived from the original and the refined antecedents. Let A be the antecedent we want to refine. Let A_{org} be the original antecedent written by the user. Let V_{new} be a set of symbolic variables so that $V \cap V_{new} = \emptyset$. Let PI_{ref} be the set of inputs at specific times, selected for refinement. Let A_{new} be a refinement of A over $V \cup V_{new}$, where A_{new} is received from A by attaching to each input $(in, t) \in PI_{ref}$ a unique variable $v_{in,t} \in V_{new}$ and adding conditions to A as follows: $A_{new} = A \wedge \bigwedge_{(in,t) \in PI_{ref}} N^t(p \rightarrow (in \text{ is } v_{in,t}))$, where $p = \neg q$ if (in, t) has a constraint $N^t(q \rightarrow (in \text{ is } e))$ in A_{org} for some Boolean expressions q and e over V, and $p = 1$ otherwise ((in, t) has no constraint in A_{org}). The reason we consider A_{org} is to avoid a contradiction between the added constraints and the original ones, due to constraints in A_{org} of the form $q \rightarrow f$.

Let $\pi^{A_{new}}$ be the defining trajectory of M and A_{new}, over $V \cup V_{new}$. Let ϕ be an assignment to V. Then $runs(A_{new}, M, \phi)$ denotes the set of all concrete trajectories π for which there is an assignment ϕ' to V_{new} so that $(\phi \cup \phi')(\pi^{A_{new}})$ is an abstraction of π. Since for all concrete trajectories π, $[(\phi \cup \phi'), \pi \models A_{new}] = 1 \iff \pi \sqsubseteq (\phi \cup \phi')(\pi^{A_{new}})$, we get that $runs(A_{new}, M, \phi)$ are exactly those π for which there is ϕ' so that $[(\phi \cup \phi'), \pi \models A_{new}] = 1$.

Theorem 1. *1. For all assignments ϕ to V, $runs(A, M, \phi) = runs(A_{new}, M, \phi)$.*

2. If $[\pi^{A_{new}} \models C] = 1$ then $\forall \phi$ it holds that $\forall \pi \in runs(A, M, \phi) : [\phi, \pi \models C] = 1$.

3. If there is ϕ' to V_{new} and $\pi \in runs(A_{new}, M, \phi \cup \phi')$ so that $[(\phi \cup \phi'), \pi \models A_{new}] = 1$ but $[(\phi \cup \phi'), \pi \models C] = 0$ then $\pi \in runs(A, M, \phi)$ and $[\phi, \pi \models A] = 1$ and $[\phi, \pi \models C] = 0$.

Theorem 1 implies that if $A_{new} \Longrightarrow C$ holds on all concrete trajectories of M, then so does $A \Longrightarrow C$. Moreover, if $A_{new} \Longrightarrow C$ yields a concrete counterexample ce, then ce is also a concrete counterexample w.r.t $A \Longrightarrow C$.

4 Selecting Inputs for Refinement

In this section we describe how exactly the refinement process is performed. We assume that $[\pi^A \models C] = X$, and thus automatic refinement is activated. Our goal is to add a small number of constraints to A forcing inputs to the value of fresh symbolic variables, while eliminating as many assignments ϕ as possible so that $[\phi, \pi^A \models C] = X$. The refinement process is incremental - inputs (in, t) that are switched from X to a fresh symbolic variable will not be reduced to X in subsequent iterations.

Choosing our refinement goal. Assume that $[\pi^A \models C] = X$, and the symbolic incomplete trace is generated. This trace contains all assignments ϕ for which $[\phi, \pi^A \models C] = X$. For each such assignment ϕ, the trajectory $\phi(\pi^A)$ is called an ***incomplete trajectory***. In addition, this trace may contain multiple nodes that are required by C to a definite value (either 0 or 1) for some assignment ϕ, but equal X. We refer to such nodes as ***undecided nodes***. We want to keep the number of added constraints small. Therefore, we choose to eliminate one undecided node (n, t) in each refinement iteration, since different nodes may depend on different inputs. A motivation for eliminating only part of the undecided nodes is that an eliminated X value may be replaced in the next iteration with a definite value that contradicts the required value (a counterexample). We suggest to choose an undecided node (n, t) with minimal number of inputs in its BCOI. Out of those, we choose a node with minimal number of nodes in its BCOI. Our experimental results support this choice. The chosen undecided node is our ***refinement goal*** and is denoted $(root, tt)$. We also choose to eliminate at once all incomplete trajectories in which $(root, tt)$ is undecided. These trajectories are likely to be eliminated by similar sets of inputs. Thus, by considering them all at once we can considerably reduce the number of refinement iterations, without adding too many variables.

The Boolean expression $(\neg f^1_{root,tt} \wedge \neg f^0_{root,tt} \wedge (g^1_{root,tt} \vee g^0_{root,tt}))$ represents the set of all ϕ for which $(root, tt)$ is undecided in $\phi(\pi^A)$. Our goal is to add a small number of constraints to A so that $(root, tt)$ will not be X whenever $(g^1_{root,tt} \vee g^0_{root,tt})$ holds.

Eliminating irrelevant inputs. Once we have a refinement goal $(root, tt)$, we need to choose inputs (in, t) for which constraints will be added to A. Naturally, only inputs in the BCOI of $(root, tt)$ are considered, but some of these inputs can be safely eliminated.

Consider an input (in, t), an assignment ϕ to V and the defining trajectory π^A. We say that (in, t) is ***relevant*** to $(root, tt)$ under ϕ, if there is a path of nodes P from (in, t) to $(root, tt)$ in M, so that for all nodes (n, t') in P, $\phi(\pi^A)(t')(n) = X$. (in, t) is ***relevant*** to $(root, tt)$ if there exists ϕ so that (in, t) is relevant to $(root, tt)$ under ϕ.

For each (in, t), we compute the set of assignments to V for which (in, t) is relevant to $(root, tt)$. The computation is performed recursively starting from $(root, tt)$. $(root, tt)$ is relevant when it is X and is required to have a definite value: $(\neg f^1_{root,tt} \wedge \neg f^0_{root,tt} \wedge (g^1_{root,tt} \vee g^0_{root,tt}))$. A source node (n, t) of $(root, tt)$ is relevant whenever $(root, tt)$ is relevant and (n, t) equals X. Let $out(n, t)$ return the sink nodes of (n, t) that are in the BCOI of $(root, tt)$. Proceeding recursively, we compute for each node (n, t) the set of assignments relevant$_{n,t}$ given by the Boolean expression $(\bigvee_{(m,t') \in out(n,t)} \text{relevant}_{m,t'}) \wedge \neg f^0_{n,t} \wedge \neg f^1_{n,t}$, until we reach the input nodes (in, t).

For all ϕ that are not in relevant$_{in,t}$, changing (in, t) from X to 0 or to 1 in $\phi(\pi^A)$ can never change the value of $(root, tt)$ in $\phi(\pi^A)$ from X to 0 or to 1. Thus, if (in, t) is chosen for refinement, a possible optimization is to constrain it to a fresh symbolic variable only when relevant$_{in,t}$ holds, as follows: relevant$_{in,t} \rightarrow \mathbf{N^t}(in \text{ is } v_{in,t})$. If (in, t) is chosen in a subsequent iteration for refinement of a new refinement goal $(root', tt')$, then the previous constraint is extended by disjunction to include the condition under which (in, t) is relevant to $(root', tt')$. Theorem 1 holds also for the optimized refinement. Let PI be the set of inputs of M. The set of all inputs that are relevant to $(root, tt)$ is $PI_{(root,tt)} \equiv \{(in, t) \mid in \in PI \wedge \text{relevant}_{in,t} \not\equiv 0\}$. Adding constraints to A for all relevant inputs (in, t) will result in a refined antecedent A_{new}. In $\pi^{A_{new}}$, it is guaranteed that $(root, tt)$ will not be undecided. Note that $PI_{(root,tt)}$ is sufficient but not minimal for elimination of all undesired X values from $(root, tt)$. Namely, adding constraints for all inputs in $PI_{(root,tt)}$ will eliminate all cases in which $(root, tt)$ is undecided. However, adding constraints for only a subset of $PI_{(root,tt)}$ may still eliminate all such cases. The set $PI_{(root,tt)}$ may be valuable to the user even if automatic refinement does not take place, since it excludes inputs that are in the BCOI of $(root, tt)$ but will not change the verification results w.r.t $(root, tt)$.

Heuristics for Selection of Important Inputs. We now propose heuristics for selecting a subset of $PI_{(root,tt)}$ for refinement. A motivation for this is that a 1 or 0 truth value may be reached even without adding constraints for all relevant inputs.

We apply the following heuristics: each node (n, t) selects a subset of $PI_{(root,tt)}$ as candidates for refinement. The final set of inputs for refinement is selected out of the candidates of $(root, tt)$. Each input in $PI_{(root,tt)}$ selects itself as a candidate. Other inputs have no candidates for refinement. $sourceCand_{n,t}$ denotes the sets of candidates of the source nodes of a node (n, t), excluding the source nodes that do not have candidates. The candidates of (n, t) are determined as follows:

1. If there are candidate inputs that appear in all sets of $sourceCand_{n,t}$, then they are the candidates of (n, t).
2. Otherwise, if (n, t) has source nodes that can be classified as control and data, then the candidates of (n, t) are the union of the candidates of its control source nodes, if this union is not empty. For example, a latch has one data source node and at least one control source node - its clock. The identity of control source nodes is automatically extracted from the netlist representation of the circuit.
3. If none of the above holds, then the candidates of (n, t) are the inputs with the largest number of occurrences in $sourceCand_{n,t}$.

We prefer to refine inputs that affect control before those that affect data since the value of control inputs has usually more affect on the verification result. Moreover, the control inputs determine when data is sampled. Therefore, if the value of a data input is required for verification, it can be restricted according to the value of previously refined control inputs. In the final set of candidates, sets of nodes that are entries of the same vector are treated as one candidate. Since the heuristics could not prefer one entry of the vector over the other, then probably only their joint value can change the verification result. Additional heuristics choose a fixed number of l candidates out of the final set.

5 Detecting Vacuity and Spurious Counterexamples

In this section we raise the problem of hidden vacuity and spurious counterexamples that may occur in STE. This problem was never addressed before in the context of STE.

In STE, A functions both as determining the level of the abstraction of M, and as determining the trajectories of M on which C is required to hold. An important point is that the constraints imposed by A are applied (using the $\sqcap$ operator) to *abstract* trajectories of M. If for some node (n, t) and assignment ϕ to V, there is a contradiction between $\phi(\sigma^A)(t)(n)$ and the value propagated through M to (n, t), then $\phi(\pi^A)(t)(n) = \perp$, indicating that there is no concrete trajectory π so that $[\phi, \pi \models A] = 1$.

In this section we point out that due to the abstraction in STE, it is possible that for some assignment ϕ to V, there are no concrete trajectories π so that $[\phi, \pi \models A] = 1$, but still $\phi(\pi^A)$ does not contain $\perp$ values. This is due to the fact that an abstract trajectory may represent more concrete trajectories than the ones that actually exist in M. Consequently, it may be that $[\phi, \pi^A \models C] \in \{1, 0\}$, and there is no indication that this result is vacuous, i.e., for all concrete trajectories π, $[\phi, \pi \models A] = 0$. Note that this problem may only happen if A contains constraints on internal nodes of M. Given a constraint a on an input, there always exists a concrete trajectory that satisfies a (unless a itself is a contradiction, which can be easily detected). This problem exists also in STE implementations that do not compute π^A, such as [8].

Example 2. We return to Example 1 from Section 2. Note that the defining trajectory π^A does not contain $\perp$. In addition, $[\pi^A \models C] = 0$ due to the assignments to V in which $v_1 = 0$. However, A never holds on concrete trajectories of M when $v_1 = 0$, since N3 at time 0 will not be equal to 1. Thus, the counterexample is spurious, but we have no indication of this fact. The problem occurs when calculating the value of (N3,0) by computing $X \sqcap 1 = 1$. If A had contained a constraint on the value of In2 at time 0, say (In2 is v_2), then the value of (N3,0) in π^A would have been $(v_1 \wedge v_2) \sqcap 1 = (v_1 \wedge v_2 ? 1 : \perp)$, indicating that for all assignments in which $v_1 = 0$ or $v_2 = 0$, π^A does not correspond to any concrete trajectory of M.

Vacuity may also occur if for some ϕ to V, C under ϕ imposes no requirements. This is due to constraints of the form $p \rightarrow f$ where $\phi(p)$ is 0.

An STE assertion $A \Longrightarrow C$ is ***vacuous*** in M if for all concrete trajectories π of M and assignments ϕ to V, either $[\phi, \pi \models A] = 0$, or C under ϕ imposes no requirements. This definition is compatible with the definition in [1] for ACTL.

We say that $A \Longrightarrow C$ ***passes vacuously*** on M if $A \Longrightarrow C$ is vacuous in M and $[\pi^A \models C] \in \{\bot, 1\}$. A counterexample π is ***spurious*** if there is no concrete trajectory π_c of M so that $\pi_c \sqsubseteq \pi$. Given π^A, the symbolic counterexample ce is ***spurious*** if for all assignments ϕ to V in ce, $\phi(\pi^A)$ is spurious. $A \Longrightarrow C$ ***fails vacuously*** on M if $[\pi^A \models C] = 0$ and ce is spurious.

As explained before, vacuity detection is required only when A constrains internal nodes. It is performed only if $[\pi^A \models C] \in \{0, 1\}$ (if $[\pi^A \models C] = \bot$ then surely $A \Longrightarrow C$ passes vacuously). In order to detect non-vacuous results in STE, we need to check whether there exists an assignment ϕ to V and a concrete trajectory π of M so that C under ϕ imposes some requirement and $[\phi, \pi \models A] = 1$. In case $[\pi^A \models C] = 0$, we also require that $[\phi, \pi \models C] = 0$. Since A can be expressed as an LTL formula, we can translate A and M into a Bounded Model Checking (BMC) [2] problem. Note that in this BMC problem we search for a satisfying assignment for A, not for its negation. Additional constraints should be added to the BMC formula as follows.

For detection of vacuous pass, the BMC formula is constrained as follows: Recall that $(g^1_{n,t}, g^0_{n,t})$ denotes the dual rail representation of (n, t) in σ^C. The Boolean expression $g^1_{n,t} \vee g^0_{n,t}$ represents all assignments ϕ to V under which C imposes a requirement on (n, t). Thus, $\bigvee_{(n,t)\in C} g^1_{n,t} \vee g^0_{n,t}$ represents all assignments ϕ under which C imposes some requirement, and is added as an additional constraint to the BMC formula. A satisfying assignment to the resulting formula constitutes a witness for $A \Longrightarrow C$.

For detection of vacuous fail, the BMC formula is constrained by conjunction with the symbolic counterexample $ce = \bigvee_{(n,t)\in C}((g^1_{n,t} \wedge \neg f^1_{n,t} \wedge f^0_{n,t}) \vee (g^0_{n,t} \wedge f^1_{n,t} \wedge \neg f^0_{n,t}))$. ce represents all assignments ϕ for which $[\phi, \pi^A \models C] = 0$. A satisfying assignment to the resulting formula constitutes a concrete counterexample for $A \Longrightarrow C$.

If BMC finds a satisfying assignment to the resulting formula, then the original truth value of $[\pi^A \models C]$ is returned. Otherwise, we conclude that the STE result is vacuous. In [13], we suggest an alternative vacuity detection algorithm that uses STE and present an additional vacuity problem that arises in constraint-based STE [8].

6 Experimental Results

We implemented our automatic refinement algorithm **AutoSTE** on top of STE in Intel's FORTE environment [12]. **AutoSTE** receives a circuit M and an STE assertion $A \Longrightarrow C$. When $[\pi^A \models C] = X$, it chooses a refinement goal $(root, tt)$ out of the undecided nodes, as described in Section 4. Next, it computes the set of relevant inputs (in, t). The Heuristics described in Section 4 are applied in order to choose a subset of those inputs. In our experimental results we restrict the number of refined candidates in each iteration to 1. A is changed as described in Section 4 and STE is rerun on the new assertion.

We ran **AutoSTE** on two different circuits, which are challenging for Model Checking: the Content Addressable Memory (CAM) from Intel's GSTE tutorial, and IBM's Calculator 2 design [14]. The latter has a complex specification.Therefore, it constitutes a good example for the benefit the user can gain from automatic refinement in STE. All runs were performed on a 3.2 GHz Pentium 4 computer with 4 GB memory.

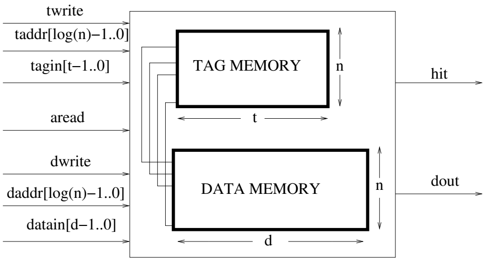

Fig. 4. Content Addressable Memory. Tag size=t, Number of entries=n, Data size=d

Content Addressable Memory. The CAM shown in Figure 4 contains 16 entries, has a data size of 64 bits and a tag size of 8 bits. It contains 1152 latches, 83 inputs and 5064 combinational gates. CAMs use bit fields called tags to identify particular data entries stored in an array. The associative read operation (aread) of CAMs consists of searching in parallel all tags in the CAM tag memory to find a match to an input tag (tagin). If a match is found, the CAM outputs the associated data entry to dout. The verification of the aread operation using STE is described in [7]. The assertions in [7] contain assumptions on the internal state of the tag memory. The user may want to check the aread operation after a write operation to the tag memory. In STE such cases can be checked by bounding the time that passed between the writing and the reading of the tag. We present the results of **AutoSTE** on 3 such assertions. Figure 5 reports the final result, number of refinement iterations, run-time in seconds and peak BDD nodes for each assertion. Table 1 reports the refinement goal and added constraint in each refinement iteration. $v_{n,t}$ denotes a fresh symbolic variable for node (n, t). $\overrightarrow{v}_{n,t}$ denotes a vector of fresh symbolic variables for a vector of nodes (n, t).

Assertion	result	Total Iter.	Time	BDD Nodes
1	pass	2	3	4768
2	fail	7	20	57424
3	fail	3	17	29006

Fig. 5. Automatic Refinement Performance on CAM Assertions

Assertion 1 checks that if a tag value $\overrightarrow{\text{TAG}}$ is written to an address $\overrightarrow{A}$ in the tag memory at time 0 (where $\overrightarrow{\text{TAG}}$ and $\overrightarrow{A}$ are vectors of symbolic variables over $\{0, 1\}$), and at time 1 $\overrightarrow{\text{TAG}}$ is read, then it should be found in the tag memory and hit should be 1: (tagin is $\overrightarrow{\text{TAG}}$) $\wedge$ (taddr is $\overrightarrow{A}$) $\wedge$ (twrite is 1) $\wedge$ **N** ((aread is 1) $\wedge$ (tagin is $\overrightarrow{\text{TAG}}$)) $\Longrightarrow$ **N** (hit is 1). Assertion 1 should pass: if at time 1 there is no write operation to the tag memory (twrite is 0), then $\overrightarrow{\text{TAG}}$ should be found in address $\overrightarrow{A}$. If at time 1 twrite is 1, $\overrightarrow{\text{TAG}}$ should be found since it is written again to the tag memory. However, $[\pi^A \models C] = X$. Since twrite and taddr at time 1 are X, the CAM cannot determine whether to write the value of tagin at time 1 to the tag memory, and to which tag entry to write it. As a result, the entire tag memory at time 1 is X.Thus, hit at time 1 is X.

After two refinements, **AutoSTE** returns a pass result. Note that only constraints necessary for obtaining the pass result were added. $\overrightarrow{\text{TAG}} \neq 0$ appears in the constraint since in this CAM implementation, the default value of the data source nodes of the tag memory is 0. Thus, when $\overrightarrow{\text{TAG}} = 0$, even without knowing if and to which entry a tag is written at time 1, the CAM determines that a tag that equals 0 exists in the tag memory.

Assertion 2 is an extension of Assertion 1. We add a constraint to the antecedent that at time 0, datamem$[\overrightarrow{A}]$ is $\overrightarrow{D}$. We also add a requirement to the consequent that at time 1, dout is $\overrightarrow{D}$. The first two refinements are the same as for assertion 1. The next

Assertion	Iteration	Goal	Added Constraint
1,2	1	hit,1	$\mathbf{N}(\overrightarrow{\text{TAG}} \neq 0 \rightarrow \text{twrite is } v_{twrite,1})$
1,2	2	hit,1	$\mathbf{N}((\overrightarrow{\text{TAG}} \neq 0 \wedge v_{twrite,1} = 1) \rightarrow \text{taddr is } \overrightarrow{v}_{taddr,1})$
2	3	dout[0],1	$\mathbf{N}(\overrightarrow{\text{TAG}} = 0 \rightarrow \text{twrite is } v_{twrite,1})$
2	4	dout[0],1	$\mathbf{N}((\overrightarrow{\text{TAG}} = 0 \wedge v_{twrite,1} = 1) \rightarrow \overrightarrow{\text{taddr}} \text{ is } \overrightarrow{v}_{taddr,1})$
2	5	dout[0],1	$\mathbf{N}(\text{dwrite is } v_{dwrite,1})$
2	6	dout[0],1	$\mathbf{N}(v_{dwrite,1} = 1 \rightarrow \overrightarrow{\text{daddr}} \text{ is } \overrightarrow{v}_{daddr,1})$
2	7	dout[0],1	$\mathbf{N}(((v_{dwrite,1} = 1) \wedge (\overrightarrow{v}_{daddr,1} = \overrightarrow{A})) \rightarrow din[0] \text{ is } v_{din[0],1})$
3	1	dout[0],2	$D[0] \neq 0 \rightarrow \text{dwrite is } v_{dwrite,0}$
3	2	dout[0],2	$(D[0] \neq 0 \wedge v_{dwrite,0} = 1) \rightarrow \overrightarrow{\text{daddr}} \text{ is } \overrightarrow{v}_{daddr,0}$
3	3	dout[0],2	$(D[0] \neq 0 \wedge \overrightarrow{A} \neq 0) \rightarrow \text{tagmem0 is } \overrightarrow{v}_{tagmem0,0}$

Table 1. Automatic Refinement of CAM Assertions

refinement goal is dout[0]. In iterations 3-4, twrite and taddr at time 1 are added to A when $\overrightarrow{\text{TAG}} = 0$, since they are required in order to determine the value of dout[0] at time 1. The relevant inputs for refinement in iterations 5-7 were dwrite, daddr and din[0], all at times 0 and 1, the initial values of all tag memory entries and of bit number 0 of all data memory entries. The final iteration yields a counterexample in which dwrite at time 1 equals 1, daddr at time 1 equals taddr at time 0, and din[0] at time 1 is different from D[0]. This counterexample stems from an erroneous specification. If new data is written at time 1 to the data entry associated with $\overrightarrow{\text{TAG}}$, then dout at time 1 will be equal to the new data. Note that only constraints relevant to this counterexample were added.

Assertion 3 is as follows: (tagin is $\overrightarrow{TAG}$)$\wedge$(taddr is $\overrightarrow{A}$)$\wedge$(twrite is 1)$\wedge$(datamem$[\overrightarrow{A}]$ is $\overrightarrow{D}$) $\wedge\mathbf{N}$((twrite is 0)$\wedge$(dwrite is 0))$\wedge\mathbf{N}^2$((aread is 1)$\wedge$(tagin is $\overrightarrow{TAG}$)$\wedge$(twrite is 0)$\wedge$ (dwrite is 0)) $\Longrightarrow$ $\mathbf{N}^2$((hit is 1) $\wedge$ (dout is $\overrightarrow{D}$)). This assertion should fail since the tag memory may already hold at time 0 a tag that equals $\overrightarrow{\text{TAG}}$. Though usually it is assumed that the CAM environment will not write the same tag to two different entries, most CAM implementations do not assume so. **AutoSTE** generates a counterexample after 3 refinement iterations. In the counterexample, tag entry 0 equals $\overrightarrow{\text{TAG}}$, and the address $\overrightarrow{A}$ to which $\overrightarrow{\text{TAG}}$ is written is different from 0. The data associated with tag entry 0 appears in dout, rather than the one written to address $\overrightarrow{A}$. This assertion demonstrates the case in which there is a need for refinement of initial values of latches (tagmem0 at time 0). Since our heuristics prefer inputs that influence control, the constraint on tagmem0 was added after constraints were added on dwrite and $\overrightarrow{\text{daddr}}$ at time 0.

Calculator Design. Calculator 2 design [14] shown in Figure 6 is used as a case study design in simulation based verification. It contains 2781 latches, 157 inputs and 56960 combinational gates. The calculator supports 4 types of commands: add, sub, shift right and shift left. *none* stands for no command. Any other command is invalid. It has two internal arithmetic pipelines: one for add/sub and one for shifts. The first argument of the command is sent at the same cycle as the command. The second argument is sent in the next cycle. The tag is a unique identifer for each of the commands from each of the 4 ports. It is sent at the same cycle as the command. The commands may be executed out of order. However, commands from the same port that use the same pipeline must return in order. The response is 1 for good, 2 for underflow, overflow or invalid command, 3 for an internal error and 0 for no response. Reset is 1 for the first 3 cycles.

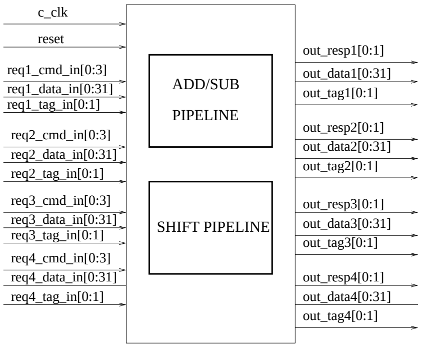

Fig. 6. Calculator

We present the results of **AutoSTE** on 4 assertions. Figure 7 reports the final result, number of refinement iterations, run-time in seconds and peak BDD nodes for each assertion. For lack of space, the description of assertion 4 exists in [13]. Table 2 reports the refinement goal and added constraint in each refinement iteration for assertions 1-3.

Assertion 1 checks whether after reset, if a port sends an add or sub command, and the other ports send no command or a command other than add and sub, then the port that sent the add/sub command receives a good response with the appropriate tag at the first available time (4 cycles after the commands were sent). A vector $\overrightarrow{P}$ of symbolic variables is used to determine which port is sending the add or sub command.

In the counterexample, a data overflow occurs for an add command sent by port 1, which triggers an invalid response at cycle 7. The BCOI of out_resp1[0] contains all command, tag and data inputs of all ports at different times. However, the set of relevant inputs contains only all entries of req1_data_in at cycles 3 and 4. req1_data_in[31] at cycles 3 and 4 is the minimal subset that is suffice to produce a counterexample, and is indeed the one chosen by our heuristics.

Assertion 2 constrains the command sent by port i to add. The msb bits of the sent data are constrained to 0 to avoid a possible overflow. The requirement is that the output data for port i should match the expected data. No constraints exist on the commands sent by other ports. In the counterexample, both ports 1 and 2 send an add command. Port 1 is answered before port 2. The assertion fails due to an erroneous specification: since port 1 has priority over port 2, port 2 may not receive a response at the first possible cycle. Due to the implementation of the priority queue, the value of an additional port had to be definite. The BCOI of (out_resp2[0],7) contains cmd, data and tag inputs of all ports at cycles 3 and 4. Out of them, only the cmd and data inputs are relevant inputs.

Assertion	result	Total Iter.	Time	BDD Nodes
1	fail	2	87	6241
2	fail	2	100	20134
3	fail	1	220	530733
4	pass	11	494	17323

Fig. 7. Automatic Refinement Performance on Calculator Assertions

Assertion 3 presents the following constraints: after reset, a port sends an add or sub command, followed by an add command with a certain tag and data arguments, while limiting the msb of the data to 0 to avoid a possible overflow. All other ports do not send an add or sub command during this time. The requirements are: the port that sent the add command receives a response with the appropriate tag value and expected output data. There was one refinement iteration. The BCOI of resp_out1[0] includes all data and tag inputs of all ports. However, only the tags of all ports at cycles 3-5 are relevant inputs. Our heuristics chose the tag of port 1 at cycle 3. Choosing any other input would require additional iterations in order to produce a counterexample. In the counterexample, the tag values of port 1 at cycles 3 and 4 are not consecutive.

Assert.	Iteration	Goal	Added Constraint
1	1	out_resp1[0],7	$\mathbf{N}^3\overrightarrow{P} = 1 \rightarrow$ req1_data_in[31] is $v_{req1\_data\_in[31],3}$
1	2	out_resp1[0],7	$\mathbf{N}^4\overrightarrow{P} = 1 \rightarrow$ req1_data_in[31] is $v_{req1\_data\_in[31],4}$
2	1	out_resp2[0],7	$\mathbf{N}^3\overrightarrow{P} = 2 \rightarrow$ req1_cmd_in is $\overrightarrow{v}_{req1\_cmd\_in,3}$
2	2	out_resp2[0],7	$\mathbf{N}^3(\overrightarrow{P} = 2 \wedge \overrightarrow{v}_{req1\_cmd\_in,3} = (add \vee sub)) \rightarrow$ req3_cmd_in is $\overrightarrow{v}_{req3\_cmd\_in,3}$
3	1	out_resp1[0],9	$\mathbf{N}^3\overrightarrow{P} = 1 \rightarrow$ req1_tag_in is $\overrightarrow{v}_{req1\_tag\_in,3}$

Table 2. Automatic Refinement of Calculator Assertions

This counterexample stems from a planted design bug documented in [14]. There is supposed to be no restriction on tag ordering. However, commands whose tags are out of order are treated as invalid.

Acknowledgement. We thank Eli Singerman for introducing us to STE and to the Forte environment.

References

1. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *CAV*, 1997.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.
3. C-T. Chou. The mathematical foundation of symbolic trajectory evaluation. In *CAV*, 1999.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
5. S. Hazelhurst and C.-J. H. Seger. Model checking lattices: Using and reasoning about information orders for abstraction. *Logic journal of IGPL*, 7(3), 1999.
6. R. P. Kurshan. *Computer-Aided Verification of coordinating processes - the automata theoretic approach*. 1994.
7. M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *DAC*, 1997.
8. J.-W. Roorda and K. Claessen. A new SAT-based algorithm for symbolic trajectory evaluation. In *CHARME*, 2005.
9. J.-W. Roorda and K. Claessen. SAT-based assistance in abstraction refinement for symbolic trajectory evaluation. In *CAV*, 2006.
10. T. Schubert. High level formal verification of next-generation microprocessors. In *DAC'03*.
11. C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2), 1995.
12. C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. F. Melham, M. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(9), 2005.
13. R. Tzoref. Automatic refinement and vacuity detection for symbolic trajectory evaluation. Master's thesis, Department of Computer Science, Technion, Israel, 2006.
14. B. Wile, W. Roesner, and J. Goss. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan-Kaufmann, 2005.
15. J.C. Wilson. *Symbolic Simulation Using Automatic Abstraction of Internal Node Values*. PhD thesis, Stanford University, Dept. of Electrical Engineering, 2001.
16. J. Yang, R. Gil, and E. Singerman. satGSTE: Combining the abstraction of GSTE with the capacity of a SAT solver. In *DCC*, 2004.
17. J. Yang and A. Goel. GSTE through a case study. In *ICCAD*, 2002.
18. J. Yang and C.-J. H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In *FMCAD*, 2002.
19. J. Yang and C.-J. H. Seger. Introduction to generalized symbolic trajectory evaluation. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(3), 2003.