Symbolic Trajectory Evaluation (STE): Automatic Refinement and Vacuity Detection

Orna Grumberg

Abstract.

Symbolic Trajectory Evaluation (STE) is a powerful technique for hardware model checking. It is based on combining 3-valued abstraction with 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 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.

Keywords. Symbolic Trajectory Evaluation (STE), model checking, abstraction-refinement, vacuity

1. Introduction

This paper is an overview of the work presented in [30] and [29]. It presents the framework of Symbolic Trajectory Evaluation (STE) and describes automatic refi nement and vacuity detection in this context.

Symbolic Trajectory Evaluation (STE) [26] is a powerful technique for hardware model checking. STE combines 3-valued abstraction with symbolic simulation. It is applied to a circuit M, described as a graph over *nodes* (gates and latches). Specifi cations in STE consist 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). For each node, STE computes a symbolic representation, often in the form of a Binary Decision Diagram (BDD) [8]. The BDD represents the value of the node as a function of the values of the circuit's inputs. For precise symbolic representation, memory requirements might be prohibitively high. Thus, in order to handle very large circuits, it is necessary to apply some form of abstraction.

Abstraction in STE is derived from the specifi cation by initializing all inputs not appearing in A to the X ("unknown") value. The rest of the inputs are initialized according to constraints in A to the values 0 or 1 or to symbolic variables. A fourth value, $\perp$, is used in STE for representing a contradiction between a constraint in A on some node (n, t) and the actual value of node n at time t in the circuit.

In [18], 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 circuit nodes. Thus, STE assertions may get one of these four values when checked on a circuit M. The values 0 and 1 indicate that the assertion fails or passes on M, respectively. The $\perp$ truth value indicates that no computation of M satisfies A. Thus, the STE assertion passes vacuously. The X truth value indicates that the antecedent is too coarse and underspecifies the circuit.

In the latter case a *refinement* is needed. Refinement in STE amounts to changing the assertion in order to present node values more accurately.

STE has been in active use in the hardware industry, and has been very successful in verifying huge circuits containing large data paths [27,25,34]. Its main drawback, however, is the need for manual abstraction and refinement, which can be labor-intensive.

In this work 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. 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 in accord with the user intention. Therefore, we assume that it is, and we make sure that our refined assertion passes on the concrete circuit if and only if the original assertion does. Second, bugs cannot automatically be fixed. Thus, counterexamples are analyzed by the user.

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 two algorithms for detecting these cases.

We implemented our automatic refinement technique within Intel's Forte environment [27]. 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.

The rest of the paper is organized as follows. Section 2 reviews related work. In Section 3 we give some background and basic definitions and notations. Section 4 describes the inherent limitations of automatic refinement of specifications versus manual refinement, and characterizes our proposed refinement technique. Section 5 presents heuristics for choosing a subset of inputs to be refined. Section 6 defines the vacuity problem in STE and suggests several methods for vacuity detection. Section 7 briefly summarizes experimental results of our refinement technique. Finally, Section 8 concludes and suggests directions for future research.

2. Related Work

Abstraction is a well known methodology in model checking for fighting the state explosion problem. Abstraction hides certain details of the system in order to result in a smaller model. Two types of semantics are commonly used for interpreting temporal logic formulas over an abstract model. In the two-valued semantics, a formula is either true or false in the abstract model. When the formula is true, it is guaranteed to hold for the concrete model as well. On the other hand, false result may be spurious, meaning that the result in the concrete model may not be false. In the three-valued semantics [7,28], a third truth value is introduced: the unknown truth value. With this semantics, the true and false truth values in the abstract model are guaranteed to hold also in the concrete model, whereas the unknown truth value gives no information about the truth value of the formula in the concrete model.

In both semantics, when the model checking result on the abstract model is inconclusive, the abstract model is refined by adding more details to it, making it more similar to the concrete model. This iterative process is called Abstraction-Refinement, and has been investigated thoroughly in the context of model checking [14,10,21,15,3].

The work presented in this paper is the first attempt to perform automatic refinement in the framework of STE. In [13], it is shown that the abstraction in STE is an abstract interpretation via a Galois connection. However, [13] is not concerned with refinement. In [32], an automatic abstraction-refinement for symbolic simulation is suggested. The main differences between our work and [32] is that we compute a set of sufficient inputs for refinement and that our suggested heuristics are significantly different from those proposed in [32].

Recently, two new refinement methods have been suggested. The automatic refinement presented in [12] is based on a notion of responsibility and can be combined with the method presented here. The method in [16] is applicable only in the SAT-based STE framework developed there. In [1], a method for automatic *abstraction* without refinement is suggested.

Generalized STE (GSTE) [36] is a significant extension of STE that can verify all ω-regular properties. Two manual refinement methods for GSTE are presented in [35]. In the first method, refinement is performed by changing the specification. In the second method, refinement is performed by choosing a set of nodes in the circuit, whose values and the relationship among them are always represented accurately. In [33], SAT-based STE is used to get quick feedback when debugging and refining a GSTE assertion graph. However, the debugging and refinement process itself is manual. An automatic refinement for GSTE has recently been introduced in [11].

An additional source of abstraction in STE is the fact that the constraints of A on internal nodes are propagated only forward through the circuit and through time. We do not deal with this source of abstraction. In [36], they handle this problem by the Bidirectional (G)STE algorithm, in which backward symbolic simulation is performed, and new constraints implied by the existing constraints are added to A. STE is then applied on the enhanced antecedent. Our automatic refinement can be activated at this stage.

Vacuity refers to the problem of trivially valid formulas. It was first noted in [4]. Automatic detection of vacuous pass under symbolic model checking was first proposed in [5] for a subset of the temporal logic ACTL called w-ACTL. In [5], vacuity is defined

as the case in which, given a model M and a formula ϕ, there exists a sub formula ξ of ϕ which does not affect the validity of ϕ. Thus, replacing ξ with any other formula will not change the truth value of ϕ in M. In [19,20] the work of [5] has been extended by presenting a general method for detecting vacuity for specifications in CTL*. Further extensions appear in [2,9].

In the framework of STE, vacuity, sometimes referred to as *antecedent failure*, is discussed in [18,26]. Roughly speaking, it refers to the situation in which a node is assigned with a $\perp$ value, implying that there are no concrete executions of the circuit that satisfy all the constraints in A. As a result, $A \Longrightarrow C$ is trivially satisfied. This is in fact a special case of vacuity as defined in [5]. The work presented here is the first to raise the problem of hidden vacuity, in which the formula is trivially satisfied despite the fact that no nodes are assigned with the $\perp$ value.

3. Background

3.1. Circuits

There are different levels in which hardware circuits can be modeled. We concentrate on a synchronous gate-level view of the circuit, in which the circuit is modeled by logical gates such as AND and OR and by delay elements (latches). Aspects such as timing, asynchronous clock domains, power consumption and physical layout are ignored, making the gate-level model an abstraction of the real circuit.

More formally, a circuit M consists of a set of nodes $\mathcal{N}$, connected by directed edges. A node is either an input node or an internal node. Internal nodes consist of latches and combinational nodes. Each combinational node is associated with a Boolean function. The nodes are connected by directed edges, according to the wiring of the electric circuit. 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 value of a latch at time 0 is determined by a given initial value. The *outputs* of the circuit are designated internal nodes whose values are of interest. We restrict the set of circuits so that 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).

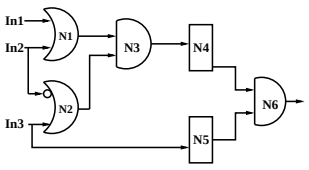

Figure 1. A Circuit

An example of a circuit is shown in Figure 1. It contains 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 clocks of the latches were omitted and we assume that 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.

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 defi ned 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.

3.2. Four-Valued Symbolic Simulation

Usually, the circuit nodes receive Boolean values, where the value of a node can be described by a Boolean expression over its inputs. In STE, a third value, X ("unknown"), is introduced. Attaching X to a certain node represents lack of information regarding the Boolean value of that node. The motivation for the introduction of X is that its use decreases the size of the Boolean expressions of the circuit nodes. This, however, is done at the expense of the possibility of receiving unknown values for the circuit outputs.

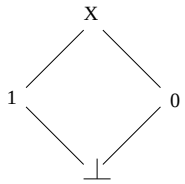

Figure 2. The $\sqsubseteq$ partial order

A fourth value, $\perp$, is also added to represent the over-constrained value, in which a node is forced both to 0 and to 1. This value indicates that a contradiction exists between external assumptions on the circuit and its actual behavior. The set of values $\mathcal{Q} \equiv \{0,1,X,\perp\}$ forms a complete lattice with the partial order $0 \sqsubseteq X$, $1 \sqsubseteq X$, $\perp \sqsubseteq 0$ and $\perp \sqsubseteq 1$ (see Figure 2 [1]. This order corresponds to set inclusion, where X represents the set $\{0,1\}$, and $\perp$ represents the empty set. As a result, the *greatest lower bound* (the lattice's meet) $\sqcap$ corresponds to set intersection and the *least upper bound* (the lattice's join) $\sqcup$ corresponds to set union. The Boolean operations AND, OR and NOT are extended to the domain $\mathcal{Q}$ as shown in Figure 3.

AND	X	0	1	$\perp$
X	X	0	X	$\perp$
0	0	0	0	$\perp$
1	X	0	1	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

OR	X	0	1	$\perp$
X	X	X	1	$\perp$
0	X	0	1	$\perp$
1	1	1	1	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

NOT	
X	X
0	1
1	0
$\perp$	$\perp$

Figure 3. 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 $s_1 \sqsubseteq s_2 \iff ((\exists n \in \mathcal{N} : s_1(n) =$

[1] Some works refer to the partial order in which X is the smallest element in the lattice and $\perp$ is the greatest.

$\perp) \vee (\forall n \in \mathcal{N} : \ s_1(n) \sqsubseteq s_2(n)))$. 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 $\sigma_1 \sqsubseteq \sigma_2 \iff ((\exists i \geq 0, n \in \mathcal{N} : \sigma_1(i)(n) = \perp) \vee (\forall\, i \geq 0 : \sigma_1(i) \sqsubseteq \sigma_2(i)))$. Note that we refer to states and sequences that contain $\perp$ values as least elements w.r.t $\sqsubseteq$.

In addition to the quaternary set of values Q, STE uses Boolean symbolic variables which enable to simulate many runs of the circuit at once. 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, $\phi(\sigma_1) \sqsubseteq \phi(\sigma_2)$.

Sequences may be incompatible with the behavior of M. A ***(symbolic) trajectory*** π is a (symbolic) sequence that is compatible with the behavior of M [24]: let $val(n,t,\pi)$ be the value of a node (n,t) as computed according to the values of its source nodes in π. It is required that for all nodes (n,t), $\pi(t)(n) \sqsubseteq val(n,t,\pi)$ (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 difference between assigning an input with a symbolic variable and assigning it with X is that a symbolic variable is used to obtain an accurate representation of the value of the input. For example, the negation of a variable v is $\neg v$ whereas the negation of X is X. In addition, if two different inputs are assigned with the same variable v in a symbolic sequence σ, then it implies that the two inputs have the same value in every concrete sequence derived from σ by applying to it an assignment ϕ. However, if the inputs are assigned with X, then it does not imply that they have the same value in any concrete sequence corresponding to σ.

Figure 4 describes a symbolic trajectory of the circuit from Figure 1 up to time 1. The values given by the user are marked in bold, and include the input values and the initial values of the latches. The notation $v_3?1:X$ stands for "if v_3 holds then 1 else X".

Time	In1	In2	In3	N1	N2	N3	N4	N5	N6
0	$\mathbf{v_1}$	**1**	$\mathbf{v_2}$	1	v_2	v_2	X	**1**	X
1	$\mathbf{v_3}$	X	**0**	$v_3?1:X$	X	X	v_2	v_2	v_2

Figure 4. Four-valued Symbolic Simulation

3.3. Trajectory Evaluation Logic (TEL)

We now describe the Trajectory Evaluation Language (TEL) used to specify properties for STE. This logic is a restricted version of the Linear Temporal Logic (LTL) [23], where only the next time temporal operator is allowed.
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 [26], i.e., f is either satisfied or not satisfied. In [18], $\mathcal{Q}$ is used also as a 4-valued truth domain for an extension of TEL. We also use a 4-valued semantics. However, our semantic definition is different from [18] w.r.t $\perp$ values. In [18], 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. The intuition behind our definition is that a sequence that contains $\perp$ value does not represent any concrete sequence, and thus vacuously satisfies all properties.
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:

$$\begin{array}{lcl}
[\phi, \sigma \models f] = \perp & \leftrightarrow & \exists i \geq 0, n \in \mathcal{N} : \phi(\sigma)(i)(n) = \perp. \text{ Otherwise:} \\
[\phi, \sigma \models n \text{ is } p] = 1 & \leftrightarrow & \phi(\sigma)(0)(n) = \phi(p) \\
[\phi, \sigma \models n \text{ is } p] = 0 & \leftrightarrow & \phi(\sigma)(0)(n) \neq \phi(p) \text{ and } \phi(\sigma)(0)(n) \in \{0, 1\} \\
[\phi, \sigma \models n \text{ is } p] = X & \leftrightarrow & \phi(\sigma)(0)(n) = X \\
[\phi, \sigma \models p \rightarrow f] & = & (\neg\phi(p) \vee \phi, \sigma \models f) \\
[\phi, \sigma \models f_1 \wedge f_2] & = & (\phi, \sigma \models f_1 \wedge \phi, \sigma \models f_2) \\
[\phi, \sigma \models \mathbf{N}f] & = & \phi, \sigma^1 \models f
\end{array}$$

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.
We define the truth value of $\sigma \models f$ as follows:

$$\begin{array}{ll}
[\sigma \models f] = 0 & \leftrightarrow \ \forall\phi : [\phi, \sigma \models f] = 0 \\
[\sigma \models f] = X & \leftrightarrow \ \forall\phi : [\phi, \sigma \models f] \neq 0 \text{ and } \exists\phi : [\phi, \sigma \models f] = X \\
[\sigma \models f] = 1 & \leftrightarrow \ \forall\phi : [\phi, \sigma \models f] \notin \{0, X\} \text{ and } \exists\phi : [\phi, \sigma \models f] = 1 \\
[\sigma \models f] = \perp & \leftrightarrow \ \forall\phi : [\phi, \sigma \models f] = \perp
\end{array}$$

It has been proved in [18] 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.

Theorem 3.1 *[29] Given a TEL formula f and two symbolic sequences σ_1 and σ_2, if $\sigma_2 \sqsubseteq \sigma_1$ then $[\sigma_2 \models f] \sqsubseteq [\sigma_1 \models f]$.*

It has been proved in [18] 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, $[\sigma \models f] \in \{1, \bot\}$ if and only if $\sigma \sqsubseteq \sigma^f$. For example, $\sigma^{q \to (n \text{ is } p)}$ is the sequence $s_{(n,q\to p)} s_x s_x s_x ...$, where $s_{(n,q\to p)}$ is the state in which n equals $(q \to p) \wedge (\neg q \to X)$, 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.

The ***defining trajectory*** π^f of M and f is a symbolic trajectory so that $[\pi^f \models f] \in \{1, \bot\}$ and for all trajectories π of M, $[\pi \models f] \in \{1, \bot\}$ if and only if $\pi \sqsubseteq \pi^f$. The $\bot$ may arise in case of a contradiction between M and f. (Similar definitions for σ^f and π^f exist in [26] with respect to a 2-valued truth domain).

Given σ^f, π^f is computed iteratively as follows: For all i, $\pi^f(i)$ is initialized to $\sigma^f(i)$. Next, the value of each node (n, i) is calculated according to its functionality and the values of its source nodes. The calculated value is then incorporated into $\pi^f(i)(n)$ using the $\sqcap$ operator. The computation of $\pi^f(i)$ continues until no new values are derived at time i. Note that since there are no combinational loops in M, it is guaranteed that eventually no new node values at time i will be derived. An example of a computation of π^f is given in Example 1.

3.4. Verification in STE

Specification in STE is given by STE assertions. 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. $M \models (A \Longrightarrow C)$ if and only if for all concrete trajectories π of M and assignments ϕ to V, $[\phi, \pi \models A] = 1$ implies that $[\phi, \pi \models C] = 1$.

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 $[\pi^A \models C] \in \{1, \bot\}$ 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 $\bot$ 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 $\bot$ values. In this paper we take the approach that supports the full semantics of STE as defined above. That is, concrete trajectories $\phi(\pi^A)$ which include $\bot$ are ignored, since they do not satisfy A and therefore vacuously satisfy $A \Longrightarrow C$.

Note that although π^A is infinite, it is sufficient to examine only a bounded prefix of length depth(A) in order to detect $\bot$ values in π^A. The first $\bot$ value in π^A is the result of the $\sqcap$ operation on some node (n, t), where the two operands have contradicting assignments 0 and 1. Since $\forall i > \text{depth}(A) : \sigma^A(i) = s_x$, it must hold that $t \leq \text{depth}(A)$.

The truth value of $\pi^A \models C$ is determined as follows:

1. If for all ϕ, there exists i, n so that $\phi(\pi^A)(i)(n) = \perp$, then $[\pi^A \models C] = \perp$.
2. Otherwise, if there exists ϕ such that for some $i \geq 0, n \in \mathcal{N}$, $\phi(\pi^A)(i)(n) \in \{0,1\}$ and $\phi(\sigma^C)(i)(n) \in \{0,1\}$, and $\phi(\pi^A)(i)(n) \neq \phi(\sigma^C)(i)(n)$, and $\phi(\pi^A)$ does not contain $\perp$, then $[\pi^A \models C] = 0$.
3. Otherwise, if there exists ϕ such that for some $i \geq 0, n \in \mathcal{N}$, $\phi(\pi^A)(i)(n) = X$ and $\phi(\sigma^C)(i)(n) \in \{0,1\}$, and $\phi(\pi^A)$ does not contain $\perp$, then $[\pi^A \models C] = X$.
4. Otherwise, $[\pi^A \models C] = 1$.

Note that, similarly to detecting $\perp$, in order to determine the truth value of $\pi^A \models C$, it is sufficient to examine only a bounded prefix of length depth(C), since $\forall i > \text{depth}(C) : \sigma^C(i) = s_x$.

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

Figure 5. The Defining Trajectory π^A

Example 1 *Consider again the circuit M in Figure 1. 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 5 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 6.*

3.5. STE Implementation

Most widely used STE implementations are BDD-based (e.g., [27]). BDDs are used to represent the value of each node (n,t) as a function of the circuit's inputs. Since node values range over the quaternary domain $\{0,1,X,\perp\}$, two BDDs are used to represent the function of each node (n,t). This representation is called ***dual rail***.

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 symbolic 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 $g^0_{n,t}$ denote 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.

The BDDs for $f^1_{n,t}$ and $f^0_{n,t}$ can be computed for each node (n,t), based on the node's functionality and the BDDs of its input nodes. Usual BDD operations are sufficient. Once this computation terminates, the BDDs for $f^1_{n,t}$, $f^0_{n,t}$ are compared with $g^1_{n,t}$, $g^0_{n,t}$ in order to determine the truth value of the specification $A \Longrightarrow C$ on M. In the following section, we further elaborate on the use of the dual rail representation in computing the STE result.

Example 2 *Consider the symbolic trajectory described in Figure 4, where* $V = \{v_1, v_2, v_3\}$.

- *The value of node* $(In1, 1)$*,* v_3*, is given by the dual rail representation* $f^1_{In1,1}(V) = v_3$*,* $f^0_{In1,1}(V) = \neg v_3$*.*
- *The value of node* $(In2, 1)$*,* X*, is given by the dual rail representation* $f^1_{In2,1}(V) = 0$*,* $f^0_{In2,1}(V) = 0$*.*
- *The value of node* $(N1, 1)$*,* $v_3?1 : X$*, is given by the dual rail representation* $f^1_{N1,1}(V) = v_3$*,* $f^0_{N1,1}(V) = 0$*.*

4. Choosing Our Automatic Refinement Methodology

Intuitively, the defi ning 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, it contains too few constraints on the values of circuit nodes. Our goal is to automatically refi ne 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 refi nement in STE. We then describe our automatic refi nement methodology, and formally state the relationship between the two abstractions, derived from the original and the refi ned antecedent. We refer only to STE implementations that compute π^A. We assume that antecedent failures are handled as described in Section 3.

We fi rst describe the handling of $\perp$ values which is required for the description of the general abstraction and refi nement process in STE. In the dual-rail notation given earlier, the Boolean expression $\neg f^1_{n,t} \vee \neg f^0_{n,t}$ represents all assignments ϕ to V for which $\phi(\pi^A)(t)(n) \neq \perp$. Thus, the Boolean expression $nbot \equiv \bigwedge_{(n,t)\in A}(\neg f^1_{n,t} \vee \neg f^0_{n,t})$ represents all assignments ϕ to V for which $\phi(\pi^A)$ does not contain $\perp$. It is suffi cient to examine only nodes (n, t) on which there exists a constraint in A. This is because there exists a node (n, t) and an assignment ϕ to V such that $\phi(\pi^A)(t)(n) = \perp$ only if there exists a node (n', t') on which there exists a constraint in A and $\phi(\pi^A)(t')(n') = \perp$. That is, the constraint on (n', t') in A contradicts the behavior of M. Thus, $[\pi^A \models C] = \perp$ if and only if $nbot \equiv 0$.

We now describe how the abstraction and refi nement process in STE is done traditionally, with the addition of supporting $\perp$ in π^A. 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:

$$\Big(\bigvee_{(n,t)\in C} ((g^1_{n,t} \wedge \neg f^1_{n,t} \wedge f^0_{n,t}) \vee (g^0_{n,t} \wedge f^1_{n,t} \wedge \neg f^0_{n,t}))\Big) \wedge nbot.$$

Each assignment ϕ in this set represents a counterexample $\phi(\pi^A)$. The counterexamples are given to the user to analyze and fi x.

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 the Boolean expression over V:

$$(\bigvee_{(n,t)\in C} ((g^1_{n,t} \vee g^0_{n,t}) \wedge \neg f^1_{n,t} \wedge \neg f^0_{n,t})) \wedge nbot.$$

The user decides how to refine the specification in order to eliminate the partial information that causes the X truth value. If $[\pi^A \models C] = \perp$, then the assertion passes vacuously. Otherwise, $[\pi^A \models C] = 1$ and the verification completes successfully.

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 in accord with the user's intention. Therefore, we assume it is, and we make sure that our refined assertion passes on the concrete circuit if and only if the original assertion does. Second, bugs cannot automatically be fixed. Thus, counterexamples are analyzed by the user.

We emphasize that automatic refinement is valuable even when it eventually results in a fail. This is because counterexamples present specific behaviors of M and are significantly easier to analyze than incomplete traces.

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 $runs(M)$ if and only if $A \Longrightarrow C$ holds on $runs(M)$.

In order to achieve the above preservation, we choose 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 5, 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. A is defined over a set of variables V. 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 for some Boolean expressions q and e over V, and $p = 1$ otherwise ((in, t) has no constraint in A). The reason we consider A is to avoid a contradiction between the added constraints and the original ones, due to constraints in A 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 $[(\phi \cup \phi'), \pi \models A_{new}] = 1$.

The reason the trajectories in $runs(A_{new}, M, \phi)$ are defined with respect to a single extension to the assignment ϕ rather than all extensions to ϕ is that we are interested in the set of all concrete trajectories that satisfy $\phi(A_{new})$ with the truth value 1. Since every trajectory $\pi \in runs(A_{new}, M, \phi)$ is concrete, it can satisfy $\phi(A_{new})$ with the truth value 1 only with respect to a single assignment to V_{new}. The fact that there are other assignments to V_{new} for which π does not satisfy $\phi(A_{new})$ with the truth value 1 is not a concern, since the truth value of $A_{new} \Longrightarrow C$ is determined only according to the concrete trajectories π and assignments ϕ to $V \cup V_{new}$ so that $[\phi, \pi \models A_{new}] = 1$.

Theorem 4.1 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 for all ϕ 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 4.1 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 $A \Longrightarrow C$. The proof of Theorem 4.1 can be found in [29].

5. Selecting Inputs for Refinement

After choosing our refinement methodology, we need to 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.

5.1. 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. Our motivation for eliminating only part of the undecided nodes is that while it is not sufficient for verification it might be sufficient for falsification. This is because an eliminated X value may be replaced in the next iteration with a definite value that contradicts the required value (a counterexample).

Algorithm 1 EliminateIrrelevantPIs((n,t)

sinks_relevant $\leftarrow \bigvee_{(m,t')\in out(n,t)}$ relevant$_{m,t'}$

relevant$_{n,t}$ $\leftarrow$ sinks_relevant $\wedge \neg f^0_{n,t} \wedge \neg f^1_{n,t}$

We suggest to choose an undecided node (n,t) with a minimal number of inputs in its BCOI. Out of those, we choose a node with a 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 refi nement iterations, without adding too many variables.

The Boolean expression $(\neg f^1_{root,tt} \wedge \neg f^0_{root,tt} \wedge (g^1_{root,tt} \vee g^0_{root,tt})) \wedge nbot$ 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.

5.2. Eliminating Irrelevant Inputs

Once we have a refi nement 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 disregarded.

Consider an input (in,t), an assignment ϕ to V and the defi ning 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 (the graph of) 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 defi nite value:

$$(\neg f^1_{root,tt} \wedge \neg f^0_{root,tt} \wedge (g^1_{root,tt} \vee g^0_{root,tt})) \wedge nbot.$$

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 as described in Algorithm 1, we compute for each (in,t) the set of assignments in which it is relevant to $(root,tt)$.

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. To see why this is true, note that if ϕ is not in relevant$_{in,t}$ it means that there is at least one node (n',t') on a path in M from (in,t) to $(root,tt)$ whose value under ϕ is defi nite (0 or 1). Since all nodes in M represent monotonic functions, changing the value of (in,t) in ϕ from X to 0 or 1 will not change the value of (n',t') and therefore will not change the value of $(root,tt)$.

Consequently, if (in,t) is chosen for refi nement, we can optimize the refi nement by associating (n,t) with a fresh symbolic variable only when relevant$_{in,t}$ holds. This can be done by adding the following constraint to the antecedent:

$\text{relevant}_{in,t} \rightarrow \mathbf{N^t}(in \text{ is } v_{in,t})$.

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 4.1 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

$$PI_{(root,tt)} = \{(in, t) \mid in \in PI \wedge \text{relevant}_{in,t} \not\equiv 0\}.$$

Adding constraints to A for all relevant inputs (in, t) will result in a refined antecedent A_{new}. In the defining trajectory of M and 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 guarantee the elimination of 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 $(root, tt)$.

5.3. Heuristics for Selection of Important Inputs

If we add constraints to A for all inputs $(in, t) \in PI_{(root,tt)}$, then we are guaranteed to eliminate all cases in which $(root, tt)$ was equal to X while it was required to have a definite value. However, such a refinement may add many symbolic variables to A, thus significantly increase the complexity of the computation of the defining trajectory. We can reduce the number of added variables at the cost of not guaranteeing the elimination of all undesired X values from $(root, tt)$, by choosing only a subset of $PI_{(root,tt)}$ for refinement. As mentioned before, a 1 or a 0 truth value may still be reached even without adding constraints for all relevant inputs.

We apply the following heuristics in order to select a subset of $PI_{(root,tt)}$ for refinement. Each node (n, t) selects a subset of $PI_{(root,tt)}$ as candidates for refinement, held in $\text{candidates}_{n,t}$. The final set of inputs for refinement is selected out of $\text{candidates}_{root,tt}$. PI denotes the set of inputs (in, t) of M. Each input in $PI_{(root,tt)}$ selects itself as a candidate. Other inputs have no candidates for refinement. Let $\text{out}(n, t)$ return the sink nodes of (n, t) that are in the BCOI of $(root, tt)$, and let $\text{degin}(n, t)$ return the number of source nodes of (n, t) that are in the BCOI of $(root, tt)$. Given a node (n, t), $\text{sourceCand}_{n,t}$ denotes the sets of candidates of the source nodes of (n, t), excluding the source nodes that do not have candidates. The candidates of (n, t) are determined according to the following conditions:

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 are candidates of most source nodes along paths from the inputs to the refinement goal, i.e., influence the refinement goal over several paths. The logic behind this heuristic is that an input that has many paths to the refinement goal is more likely to be essential to determine the value of the refinement goal than an input that has less paths to the refinement goal.

We prefer to refine inputs that affect control before those that affect data since the value of control inputs has usually more effect 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 did 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.

6. 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, the antecedent 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 $\phi(\pi^A)(t)(n) = \bot$, 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 $\bot$ 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 is possible to get $[\phi, \pi^A \models C] \in \{1, 0\}$ without any 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 [24].

Example 3 *We return to Example 1 from Section 3. Note that the defining trajectory π^A does not contain $\bot$. 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 $(v_1 \wedge v_2) \sqcap 1 = (v_1 \wedge v_2 ? 1 : \bot)$, 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 [5] for ACTL.

We say that $A \Longrightarrow C$ ***passes vacuously*** on M if $A \Longrightarrow C$ is vacuous in M and $[\pi^A \models C] \in \{\perp, 1\}$. 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. We believe that this definition is more appropriate than a definition in which ce is spurious if there exists ϕ that satisfies ce and $\phi(\pi^A)$ is spurious. The reason is that the existence of at least one non-spurious counterexample represented by ce is more interesting than the question whether each counterexample represented by ce is spurious or not.

We say that $A \Longrightarrow C$ ***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 $[\pi^A \models C] = \perp$ 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 the original STE result is fail, namely, $[\pi^A \models C] = 0$, π should also constitute a counterexample for $A \Longrightarrow C$. That is, we require that $[\phi, \pi \models C] = 0$.

We propose two different algorithms for vacuity detection. The first algorithm uses Bounded Model Checking (BMC) [6] and runs on the concrete model. The second algorithm uses STE and requires automatic refinement. The algorithm that uses STE takes advantage of the abstraction in STE, as opposed to the first algorithm which runs on the concrete model. In case non-vacuity is detected, the trajectory produced by the second algorithm (which constitutes either a witness or a counterexample) may not be concrete. However, it is guaranteed that there exists a concrete trajectory of which the produced trajectory is an abstraction. The drawback of the algorithm that uses STE, however, is that it requires automatic refinement.

6.1. Vacuity Detection using Bounded Model Checking

Since A can be expressed as an LTL formula, we can translate A and M into a Bounded Model Checking (BMC) problem. The bound of the BMC problem is determined by the depth of A. 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 in order to fulfill the additional requirements on the concrete trajectory.

For detection of vacuous pass, the BMC formula is constrained in the following way: Recall that $(g^1_{n,t}, g^0_{n,t})$ denotes the dual rail representation of the requirement on the node (n, t) in 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 to V under which C imposes some requirement. This expression is added as an additional constraint to the BMC formula. If BMC finds a satisfying assignment to the resulting formula, then the assignment of BMC to the nodes in M constitutes a witness indicating that $A \Longrightarrow C$ passed non-vacuously. Otherwise, we conclude that $A \Longrightarrow C$ passed vacuously.

For detection of vacuous fail, the BMC formula is constrained by conjunction with the (symbolic) counterexample ce. For STE implementations that compute π^A, $ce = \bigvee_{(n,t)\in C}((g^1_{n,t} \wedge \neg f^1_{n,t} \wedge f^0_{n,t}) \vee (g^0_{n,t} \wedge f^1_{n,t} \wedge \neg f^0_{n,t}))$. There is no need to add the $nbot$ constraint that guarantees that none of the nodes equals $\perp$, since the BMC formula runs on the concrete model, and thus the domain of the nodes in the BMC formula is Boolean. If BMC finds a satisfying assignment to the resulting formula, the assignment of BMC to the nodes in M constitutes a concrete counterexample for $A \Longrightarrow C$. Otherwise, we conclude that $A \Longrightarrow C$ failed vacuously.

Vacuity detection using BMC is an easier problem than solving the original STE assertion $A \Longrightarrow C$ using BMC. The BMC formula for $A \Longrightarrow C$ contains the following constraints on the values of nodes:

- The constraints of A.
- The constraints of M on nodes appearing in A.
- The constraints of M on nodes appearing in C.
- A constraint on the values of the nodes appearing in C that guarantees that at least one of the requirements in C does not hold.

On the other hand, the BMC formula for vacuity detection contains only the first two types of constraints on the values of nodes. Therefore, for vacuity detection using BMC, only the BCOI of the nodes in A is required, whereas for solving the original STE assertion $A \Longrightarrow C$ using BMC, both the BCOI of the nodes appearing in A and the BCOI of the nodes appearing in C are required.

6.2. Vacuity Detection using Symbolic Trajectory Evaluation

For vacuity detection using STE, the first step is to split A into two different TEL formulas: A^{in} is a TEL formula that contains exactly all the constraints of A on inputs, and A^{out} is a TEL formula that contains exactly all the constraints of A on internal nodes. If there exists an assignment ϕ to V so that $[\phi, \pi^{A^{in}} \models A^{out}] = 1$, then we can conclude that there exists a concrete trajectory of M that satisfies A. Note that since A^{in} does not contain constraints on internal nodes, it is guaranteed that no hidden vacuity occurs. However, it is also necessary to guarantee that in case $[\pi^A \models C] = 1$, C under ϕ imposes some requirement, and in case $[\pi^A \models C] = 0$, then $\phi(\pi^{A^{in}})$ should constitute a counterexample. Namely, $\phi \wedge ce \not\equiv 0$, where ce is the symbolic counterexample.

If we cannot find such an assignment ϕ, this does not necessarily mean that the result of $A \Longrightarrow C$ is vacuous: if there are assignments ϕ to V for which $[\phi, \pi^{A^{in}} \models A^{out}] = X$, then the trajectory $\phi(\pi^{A^{in}})$ is potentially an abstraction of a witness or a concrete counterexample for $A \Longrightarrow C$. However, it is too abstract in order to determine whether or not A^{out} holds on it. If we refine A^{in} to a new antecedent as described in Section 4, then it is possible that the new antecedent will yield more refined trajectories that contain enough information to determine whether they indeed represent a witness or concrete counterexample.

Algorithm 2 describes vacuity detection using STE. It receives the original antecedent A and consequent C. In case $[\pi^A \models C] = 0$, it also receives the symbolic counterexample ce. inputConstraints is a function that receives a TEL formula A and returns a new TEL formula that consists of the constraints of A on inputs. Similarly, internalConstraints returns a new TEL formula that consists of the constraints of A on

internal nodes. Note that since A^{in} does not contain constraints on internal nodes, then $\pi^{A^{in}}$ does not contain $\perp$ values, and thus we can assume that $f^1_{n,t} \wedge f^0_{n,t}$ never holds. By abuse of notation, $f^1_{n,t}$ and $f^0_{n,t}$ are here the dual rail representation of a node (n,t) in $\pi^{A^{in}}$. Similarly, we use $g^1_{n,t}$ and $g^0_{n,t}$ for the dual rail representation of a node (n,t) in the defining sequence of either C or A^{out}, according to the context.

Algorithm 2 STEVacuityDetection(A, C, ce)

1: $A^{in} \leftarrow$ inputConstraints(A)
2: $A^{out} \leftarrow$ internalConstraints(A)
3: $\Phi \leftarrow \bigwedge_{(n,t)\in A^{out}}((g^1_{n,t} \wedge f^1_{n,t}) \vee (g^0_{n,t} \wedge f^0_{n,t}))$
   {Φ represents all assignments to V for which $[\phi, \pi^{A^{in}} \models A^{out}] = 1$}
4: **if** $[\pi^A \models C] = 1 \wedge ((\bigvee_{(n,t)\in C}(g^1_{n,t} \vee g^0_{n,t})) \wedge \Phi) \not\equiv 0$ **then**
5: return non-vacuous
6: **else if** $[\pi^A \models C] = 0 \wedge ((\Phi \wedge ce) \not\equiv 0)$ **then**
7: return non-vacuous
8: **end if**
9: **if** $\exists \phi : [\phi, \pi^{A^{in}} \models A^{out}] = X$ **then**
10: $A^{in} \leftarrow$ refine(A^{in})
11: goto 3
12: **else**
13: return vacuous
14: **end if**

The algorithm computes the set Φ, which is the set of all assignments to V for which $[\phi, \pi^{A^{in}} \models A^{out}] = 1$. Lines 4 and 6 check whether there exists a suitable assignment ϕ in Φ that corresponds to a witness or to a counterexample. If such a ϕ exists, then the result is non-vacuous. If no such ϕ exists, then if there exist assignments for which the truth value of A^{out} on $\pi^{A^{in}}$ is X, then A^{in} is refined and Φ is recomputed. Otherwise, the result is vacuous.

Note that in case $[\pi^A \models C] = 0$, we check whether Φ contains an assignment that constitutes a counterexample by checking that the intersection between Φ and the symbolic counterexample ce produced for $[\pi^A \models C]$ is not empty. However, as a result of the refinement, Φ may contain new variables that represent new constraints of the antecedent that were not taken into account when computing ce. The reason that checking whether $(\Phi \wedge ce) \not\equiv 0$ still returns a valid result is as follows. By construction, we know that for all assignments $\phi \in \Phi$, $[\phi, \pi^{A^{in}} \models A^{out}] = 1$. Since $[\phi, \pi^{A^{in}} \models A^{in}] = 1$, we get that $[\phi, \pi^{A^{in}} \models A^{in} \cup A^{out}] = 1$, where $A^{in} \cup A^{out}$ is the TEL formula that contains exactly all the constraints in A^{in} and A^{out}. Since $[\phi, \pi^{A^{in}} \models A^{out}] = 1$, we get that $\phi(\pi^{A^{in}})$ does not contain $\perp$ values. Therefore, for all nodes (n,t) so that $\phi(\pi^A)(t)(n) = b, b \in \{0,1\}$ it holds that $\phi(\pi^{A^{in}})(t)(n) = b$. Thus, for all $\phi' \in ce$, ϕ' is a counterexample also with respect to the antecedent $A^{in} \cup A^{out}$.

Besides the need for refinement, an additional drawback of Algorithm 2 in comparison with vacuity detection using BMC, is that it attempts to solve a much harder problem - it computes a set of trajectories that constitute witnesses or concrete counterexamples, whereas in vacuity detection using BMC only one such trajectory is produced - a satisfying assignment to the SAT formula. This is in analogy to using STE versus

using BMC for model checking - STE finds the set of all counterexamples for $A \Longrightarrow C$, while BMC finds only one counterexample. However, the advantage of Algorithm 2 is that it exploits the abstraction in STE, whereas vacuity detection using BMC runs on the concrete model.

In [29], vacuity detection for SAT-based STE is presented as well.

6.3. Preprocessing for Vacuity Detection

There are some cases in which even if there exist constraints in A on internal nodes, vacuity detection can be avoided by a preliminary analysis based on the following observation: hidden vacuity may only occur if for some assignment ϕ to V, an internal node (n,t) is constrained by A to either 0, or 1, but its value as calculated according to the values of its source nodes is X. We call such a node (n,t) a *problematic node*. For example, in Example 1 from Section 3, the value of (N3,0) as calculated according to its source nodes is X, and it is constrained by A to 1.

In order to avoid unnecessary vacuity detection, we suggest to detect all problematic nodes as follows. Let $int(A)$ denote all internal nodes (n,t) on which there exists a constraint in A. Let $h^1_{n,t}$ and $h^0_{n,t}$ denote the dual rail representation of the node (n,t) in σ^A. Let $m^1_{n,t}$ and $m^0_{n,t}$ denote the dual rail representation of the value of (n,t) as calculated according to the values of its source nodes in π^A. Then the Boolean expression $\bigvee_{(n,t)\in int(A)}((h^0_{n,t} \vee h^1_{n,t}) \wedge \neg m^1_{n,t} \wedge \neg m^0_{n,t})$ represents all assignments to V for which there exists a problematic node (n,t). If this Boolean expression is identical to 0, then no problematic nodes exist and vacuity detection is unnecessary.

7. Experimental Results

We implemented our automatic refinement algorithm **AutoSTE** on top of STE in Intel's FORTE environment [27]. **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 5. Next, it computes the set of relevant inputs (in,t). The Heuristics described in Section 5 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 5 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 [31]. 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.

A detailed description of the experiments can be found in [29].

8. Conclusions and Future Work

This work describes a first attempt at automatic refinement of STE assertions. We have developed an automatic refinement technique which is based on heuristics. The refined assertion preserves the semantics of the original assertion. We have implemented our

automatic refinement in the framework of Forte, and ran it on two nontrivial circuits of dissimilar functionality. The experimental results show success in automatic verification of several nontrivial assertions.

Another important contribution of our work is identifying that STE results may hide vacuity. This possibility was never raised before. We formally defined STE vacuity and proposed two methods for vacuity detection.

Additional work is needed in order to further evaluate the suggested automatic refinement on industrial-size examples of different functionality. Such an evaluation is very likely to result in new heuristics. A preliminary work has recently been done for STE in [12] and for GSTE in [11].

We would also like to implement our suggested vacuity detection algorithms and compare their performance. In addition, we would like to develop an automatic refinement techniques to SAT based STE [33,24,17], and integrate SAT based refinement techniques [22,10].

References

[1] S. Adams, M. Bjork, T. Melham, and C. Seger. Automatic abstraction in symbolic trajectory evaluation. In *8th International Conference on Formal methods in Computer-Aided Design (FMCAD'07)*, Austin, Texas, November 2007.

[2] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. Enhanced vacuity detection in linear temporal logic. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, Boulder, CO, USA, July 2003. Springer.

[3] Sharon Barner, Daniel Geist, and Anna Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *CAV'02: Proceedings of Conference on Computer-Aided Verification*, 2002.

[4] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 596–602. ACM Press, 1994.

[5] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in ACTL formulas. In *CAV'97: Proceedings of Conference on Computer-Aided Verification*, 1997.

[6] Armin Biere, Allesandro Cimatti, Edmond M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS'99: Conference on tools and algorithms for the construction and analysis of systems*, 1999.

[7] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification*, pages 274–287, 1999.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transactions on Computers*, C-35(8):677–691, 1986.

[9] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Vardi. Regular vacuity. In *13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, Lecture Notes in Computer Science, Saarbrucken, Germany, October 2005. Springer-Verlag.

[10] Pankaj Chauhan, Edmond M. Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *FMCAD'02: Proceedings of the Forth International Conference on Formal Methods in Computer-Aided Design*, 2002.

[11] Y. Chen, Y. He, F. Xie, and J. Yang. Automatic abstraction refinement for generalized symbolic trajectory evaluation. In *8th International Conference on Formal methods in Computer-Aided Design (FMCAD'07)*, Austin, Texas, November 2007.

[12] Hana Chockler, Orna Grumberg, and Avi Yadgar. Efficient automatic ste refinement using responsibility. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, LNCS, Budapest, March-April 2008. Springer.

[13] Ching-Tsun Chou. The mathematical foundation of symbolic trajectory evaluation. In *CAV'99: Proceedings of Conference on Computer-Aided Verification*, 1999.

[14] Edmond M. Clarke, Orna Grumberg, S. Jha, Y. Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV'00: Proceedings of Conference on Computer-Aided Verifi cation*, 2000.

[15] Edmond M. Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *CAV'02: Proceedings of Conference on Computer-Aided Verifi cation*, 2002.

[16] Orna Grumberg, Assaf Schuster, and Avi Yadgar. 3-valued circuit SAT for STE with automatic refinement. In *Fifth International Symposium on Automated Technology for Verifi cation and Analysis (ATVA'07)*, volume 4762 of *LNCS*, Tokyo, Japan, October 2007.

[17] Orna Grumberg, Assaf Schuster, and Avi Yadgar. 3-Valued Circuit SAT for STE with Automatic Refinement . In *ATVA '07*, 2007.

[18] Scott Hazelhurst and Carl-Johan H. Seger. Model checking lattices: Using and reasoning about information orders for abstraction. *Logic journal of IGPL*, 7(3), 1999.

[19] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. In *CHARME'99: Conference on Correct Hardware Design and Verifi cation Methods*, pages 82–96, 1999.

[20] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. *Software Tools for Technology Transfer*, 4(2), 2003.

[21] R. Kurshan. *Computer-Aided Verifi cation of Coordinating Processes*. Princeton Univ. Press, 1994.

[22] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS'03: Conference on tools and algorithms for the construction and analysis of systems*, 2003.

[23] A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science (FOCS'77)*, 1977.

[24] Jan-Willem Roorda and Koen Claessen. A new SAT-based algorithm for symbolic trajectory evaluation. In *CHARME'05: Proceedings of Correct Hardware Design and Verifi cation Methods*, 2005.

[25] Tom Schubert. High level formal verification of next-generation microprocessors. In *DAC'03: Proceedings of the 40th conference on Design automation*.

[26] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2), 1995.

[27] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom F. Melham, Mark Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(9), 2005.

[28] Sharon Shoham and Orna Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. In *CAV'03: Proceedings of Conference on Computer-Aided Verifi cation*, 2003.

[29] Rachel Tzoref. Automated refinement and vacuity detection for Symbolic Trajectory Evaluation. Master's thesis, Department of Computer Science, Technion - Israel Institute of Technology, 2006.

[30] Rachel Tzoref and Orna Grumberg. Automatic refinement and vacuity detection for Symbolic Trajectory Evaluation. In *18th International Conference on Computer Aided Verifi cation (CAV'06)*, LNCS 4144, Seattle, August 2006.

[31] Bruce Wile, Wolfgang Roesner, and John Goss. *Comprehensive Functional Verifi cation: The Complete Industry Cycle*. Morgan-Kaufmann, 2005.

[32] James C. Wilson. *Symbolic Simulation Using Automatic Abstraction of Internal Node Values*. PhD thesis, Stanford University, Dept. of Electrical Engineering, 2001.

[33] Jin Yang, Rami Gil, and Eli Singerman. satGSTE: Combining the abstraction of GSTE with the capacity of a SAT solver. In *DCC*, 2004.

[34] Jin Yang and Amit Goel. GSTE through a case study. In *ICCAD*, 2002.

[35] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In *FMCAD'02: Proceedings of the Forth International Conference on Formal Methods in Computer-Aided Design*, 2002.

[36] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(3), 2003.