Automatic Refinement and Vacuity Detection for Symbolic Trajectory Evaluation

Rachel Tzoref

Automatic Refinement and Vacuity Detection for Symbolic Trajectory Evaluation

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Rachel Tzoref

Submitted to the Senate of the Technion - Israel Institute of Technology

Sivan 5766 Haifa June 2006

The research thesis was done under the supervision of Prof. Orna Grumberg in the Department of Computer Science.

I would like to deeply thank Prof. Orna Grumberg for her devoted guidance, help and support.

The generous financial help of the Technion is gratefully acknowledged.

Contents

Abstract 1

Notations and Abbreviations 3

1 Introduction 5
- 1.1 Related Work . . . 6
- 1.2 Outline of the Thesis . . . 8

2 Background 9
- 2.1 Circuits . . . 9
- 2.2 Simulation Based Verification . . . 10
- 2.3 Three-valued Simulation . . . 11
- 2.4 Symbolic Simulation . . . 12
- 2.5 Three-valued Symbolic Simulation . . . 13
- 2.6 Circuits Values in STE . . . 13
- 2.7 States, Sequences and Trajectories . . . 14
- 2.8 Symbolic Trajectory Evaluation . . . 15

3 Choosing Our Automatic Refinement Methodology 20
- 3.1 The Relationship Between the Abstract and Refined Assertions . . 21

4 Selecting Inputs for Refinement 26
- 4.1 Choosing our refinement goal . . . 26
- 4.2 Eliminating irrelevant inputs . . . 27
- 4.3 Heuristics for Selection of Important Inputs . . . 28

5 Detecting Vacuity and Spurious Counterexamples 31
- 5.1 Vacuity Detection using Bounded Model Checking . . . 33
- 5.2 Vacuity Detection using Symbolic Trajectory Evaluation . . . 34
  - 5.2.1 Vacuity Detection using satGSTE . . . 36
- 5.3 Preliminary Stage in Vacuity Detection . . . 37

6 Experimental Results **38**
6.1 Content Addressable Memory 38
6.2 Calculator Design 41

7 Conclusions and Future Work **46**

A Additional Vacuity in Constraint-based STE **47**

Bibliography **53**

List of Figures

2.1 A Circuit 10
2.2 The $\sqsubseteq$ partial order 13

6.1 Content Addressable Memory. Tag size=t, Number of entries=n, Data size=d 39
6.2 Calculator 42

Abstract

Model checking is an efficient procedure to check whether or not a given model fulfills a desired specification. Symbolic Trajectory Evaluation (STE) is a powerful technique for model checking of hardware circuits. 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. Our work 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. Our experimental results show success in automatically identifying a set of constraints that are crucial for reaching a definite result. In addition, our work 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.

Notations and Abbreviations

STE – Symbolic Trajectory Evaluation
M – A circuit
$\mathcal{N}$ – The set of nodes of M
(n, t) – A node of M at time t
X – The unknown value
$\perp$ – The over-constrained value
GSTE – Generalized Symbolic Trajectory Evaluation
BCOI – Bounded Cone of Influence
TEL – Trajectory Evaluation Language
A – An antecedent of an STE assertion
C – A consequent of an STE assertion
$\mathbf{N}$ – The next time operator
$\mathbf{N^t}$ – The application of t next time operators
σ^f – The defining sequence of a TEL formula f
π^f – The defining trajectory of a circuit M and a TEL formula f
$f^1_{n,t}$,$f^0_{n,t}$ – The dual rail of (n, t) in π^A
$g^1_{n,t}$,$g^0_{n,t}$ – The dual rail of (n, t) in σ^C
$nbot$ – The set of assignments for which no $\perp$ values exist in π^A
ce – The symbolic counterexample
A_{org} – The original antecedent written by the user
A_{new} – The refined antecedent
$v_{n,t}$ – A fresh symbolic variable for a node (n, t)

Chapter 1

Introduction

Symbolic Trajectory Evaluation (STE) [20] is a powerful technique for hardware model checking. STE is based on combining three-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 fourth 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 node values more accurately.

STE assertions may either pass or fail on M. In [12], a four-valued truth domain $\{0, 1, X, \perp\}$ is defined for the temporal language of STE, corresponding to the four-valued domain of the values of the circuit nodes. The motivation for a four-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) [28] is a significant extension of STE that can verify all ω-regular properties. (G)STE has been in active use in industry, and has been very successful in verifying huge circuits containing large data paths [21, 19, 26]. Its main drawback, however, is the need for manual abstraction and refinement,

which can be very labor-intensive.

In this thesis 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 correct. 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.

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 several methods for detecting these cases.

We implemented our automatic refinement technique within Intel's Forte environment [21]. 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.

1.1 Related Work

Abstraction is a well known methodology in model checking for fighting the state explosion problem, in which certain details of the system are hidden in order to result in a smaller model. Two types of semantics exist for interpreting temporal logic formulas over an abstract model. In the two-valued semantics, a formula is either true or false in an abstract model. True is guaranteed to hold for the concrete model as well, whereas false may be spurious, meaning it does not guarantee that

the result in the concrete model is false as well. In the three-valued semantics [6, 22], a third truth value is introduced: the unknown truth value. 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 [9, 7, 15, 11, 1, 10].

In [8], it is shown that the abstraction in STE is an abstract interpretation via a Galois connection. However, automatic refinement has never been suggested before for STE. The work that is closest to ours is [24], which suggests an automatic abstraction-refinement for symbolic simulation. The main differences between our work and [24] is that we compute a set of sufficient inputs for refinement and that our suggested heuristics are significantly different from those proposed in [24]. Our work is the first attempt to perform automatic refinement in the framework of STE.

Two manual refinement methods for GSTE are presented in [27]. 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 [25], 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 additional source of abstraction in STE is the fact that the constraints of A on internal nodes are propagated only forwards through the circuit and through time. We do not deal with this source of abstraction. In [28], 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 by Beatty and Bryant [2]. Automatic detection of vacuous pass under symbolic model checking was first proposed in [4] for a subset of the temporal logic ACTL called w-ACTL. In [4], 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. Kuperman and Vardi [13, 14] extend the work of [4] by presenting a general method for detecting vacuity for specifications in CTL*.

In the framework of STE, vacuity, sometimes referred to as *antecedent failure*, is discussed in [12, 20]. 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 [4]. Our work 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.

1.2 Outline of the Thesis

In Chapter 2 we give some background and basic definitions and notations. Chapter 3 describes the inherent limitations of automatic refinement of specifications versus manual refinement, and characterizes our proposed refinement technique. Chapter 4 presents heuristics for choosing a subset of inputs to be refined. Chapter 5 defines the STE vacuity problem and suggests several vacuity detection methods. An additional vacuity problem is described in Appendix A. Chapter 6 presents experimental results of our refinement technique. Finally, Chapter 7 presents conclusions and future research directions.

Chapter 2

Background

2.1 Circuits

There are different levels in which hardware circuits can be modelled. We concentrate on a synchronous gate-level view of the circuit, in which the circuit is modelled 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 $\mathcal{N}$. 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. 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 set of nodes entering a certain node are its *source nodes*, and the set of nodes to which a node enters are its *sink nodes*. The value of a combinational node at time t can be expressed as a Boolean expression over its source nodes at time t. 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 source nodes of a latch can be classified as control and data. A latch has one data source node and at least one control source node - its clock. It may have other control source nodes such as set and reset. 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 this work we refer to a node n at a specific time t as a node (n, t).

An example of a circuit is shown in Figure 2.1. 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.

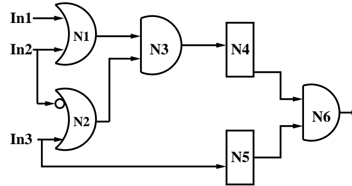

Figure 2.1: A Circuit

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). The BCOI 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.

2.2 Simulation Based Verification

The most common way to verify that a circuit implementation obeys its specification is by simulation. A *Boolean simulation test* is an assignment of Boolean values to the circuit inputs along time and to the latches at time 0. A simulator is a software or hardware tool that receives a circuit and a simulation test and performs a Boolean simulation of the circuit, i.e., calculates the value of each node n along time by computing the Boolean expression of n at time t over the values of its source nodes at time t and possibly $t-1$ (in case of a latch). Since the simulator works on a logical gate-level model of the circuit and ignores aspects such as timing and asynchronous clock domains, this calculation is an approximation of the real values of the circuit. The simulator may also receive expected results, i.e, expected values of the circuit outputs at specific times, and compare them to the actual values it calculated. Note that the verification is valid only with respect to the simulation tests provided to the simulator.

Table 2.1 describes a Boolean simulation of the circuit described in Figure 2.1. It contains the value of each node at time 0 and 1. The values given by the simulation test are marked in bold, and include the input values and the initial values of the latches.

Given a circuit with n inputs and m latches, if we want to verify that the circuit obeys its specification for all possible input values up to time t and for all possible initial values of latches, then $2^{n \cdot (t+1)+m}$ simulation tests are required. Even for

Time	In1	In2	In3	N1	N2	N3	N4	N5	N6
0	**0**	**1**	**0**	1	0	0	**0**	**1**	0
1	**1**	**1**	**1**	1	1	1	0	0	0

Table 2.1: Boolean Simulation

AND	X	0	1
X	X	0	X
0	0	0	0
1	X	0	1

OR	X	0	1
X	X	X	1
0	X	0	1
1	1	1	1

NOT	
X	X
0	1
1	0

Table 2.2: Ternary operations

medium sized circuits, this number is infeasible. Thus, simulation is performed for only a subset of possible tests. Usually, the tests are directed by a user to certain scenarios which are of interest.

2.3 Three-valued Simulation

In three-valued simulation, the simulation test may assign Boolean values to only part of the inputs at different times and latches at time 0, in which case the simulator assigns the value X to all inputs and latches that were not assigned by the simulation test. The X value represents the ”unknown” value, and is used to abstract away parts of the circuit. Attaching X to a certain node represents lack of information regarding the truth value of that node. A three-valued simulator is a simulator that can calculate the values of the nodes over the domain $\{0, 1, X\}$ by extending the Boolean operations to this domain as shown in Table 2.2.

Table 2.3 describes a three-valued simulation of the circuit from Figure 2.1 up to time 1. A subset of the inputs at times 0 and 1 and the initial values of the latches are not assigned by the simulation test.

Time	In1	In2	In3	N1	N2	N3	N4	N5	N6
0	**0**	X	**0**	X	X	X	X	X	X
1	X	X	**1**	X	1	X	X	0	0

Table 2.3: Three-valued Simulation

A motivation for using three-valued simulation is to reduce the number of simulation tests. For example, let us assume that in the simulation described in 2.3, our goal is to verify that the value of the node N6 at time 1 is 0. For the given simulation test this is indeed the case. Thus, we can conclude that the value of N6 at time 1 is 0 also for the 2^5 Boolean simulation tests implied from the given test

by replacing the unknown input values and unknown initial latches values by any combination of Boolean values. The drawback of three-valued simulation is that if too few inputs have specified values, then the circuit outputs may receive the X value, in which case it could not be determined whether or not the verification succeeded.

2.4 Symbolic Simulation

Another way to reduce the number of performed simulation tests is to use symbolic simulation. Let V be a set of symbolic Boolean variables over the domain $\{0, 1\}$. A *Boolean symbolic simulation test* is an assignment of either Boolean values (0 or 1) or of symbolic Boolean variables out of V to the inputs of the circuit and to the latches at time 0. A symbolic simulator is a simulator that receives a circuit and a symbolic simulation test, and performs symbolic simulation [3] of the circuit, i.e, the symbolic expression of each node (n, t) is computed as a function of the expressions of its source nodes. Table 2.4 describes a symbolic simulation of the circuit from Figure 2.1 up to time 1.

Time	In1	In2	In3	N1	N2	N3	N4	N5	N6
0	$\mathbf{v_1}$	**1**	$\mathbf{v_2}$	1	v_2	v_2	**0**	**1**	0
1	$\mathbf{v_3}$	$\mathbf{v_4}$	**0**	$v_3 \vee v_4$	$\neg v_4$	$v_3 \wedge \neg v_4$	v_2	v_2	v_2

Table 2.4: Symbolic Simulation

Note that a single Boolean symbolic simulation test represents multiple Boolean simulation tests, one for each assignment to the variables in V. For example, the symbolic simulation test in Table 2.4 represents 2^4 Boolean simulation tests, since it contains 4 different symbolic Boolean variables. The downside of symbolic simulation is that the size of the Boolean expressions of the circuit nodes is exponential in the number of symbolic variables which increases as the simulation progresses in time, since the value of each node n at time t may depend on inputs at all times $0 \leq t' \leq t$.

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 Boolean symbolic simulation test, then it implies that they have the same value in every Boolean simulation test derived from the symbolic simulation test. However, if the inputs are assigned with X, then it does not imply that they have the same value.

2.5 Three-valued Symbolic Simulation

Three-valued symbolic simulation combines three-valued simulation with symbolic simulation. The motivation for this combination is that the use of the X value decreases the size of the Boolean expressions of the circuit nodes, at the expense of the possibility to receive unknown values for the circuit outputs. Given a set of symbolic Boolean variables V, a three-valued symbolic expression is an expression consisting of ternary operations, applied to $V \cup \{0, 1, X\}$. A three-valued symbolic test assigns three-valued symbolic expressions to the circuit inputs along time and to the latches at time 0. A three-valued symbolic simulator receives a circuit and a three-valued symbolic test, and performs simulation of the circuit by calculating the three-valued symbolic expression of each node along time according to the symbolic expressions of its source nodes. Table 2.5 describes a three-valued symbolic simulation of the circuit from Figure 2.1 up to time 1. 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

Table 2.5: Three-valued Symbolic Simulation

2.6 Circuits Values in STE

Recall that a circuit is modelled as a set of nodes $\mathcal{N}$, connected by directed edges.

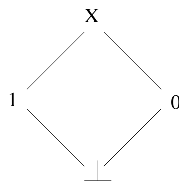

Figure 2.2: The $\sqsubseteq$ partial order

In STE, the circuit nodes receive values out of the set $\mathcal{Q} \equiv \{0, 1, X, \bot\}$. The fourth 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 a contradiction exists between external assumptions on the circuit and its actual behavior. $\mathcal{Q}$ 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 is interpreted as the set $\{0, 1\}$, and $\bot$ is interpreted as the empty set. As a result, the *greatest lower bound* $\sqcap$ corresponds

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
$\bot$	$\bot$

Table 2.6: Quaternary operations

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 Table 2.6.

2.7 States, Sequences and Trajectories

A ***state*** s of the circuit M is an assignment of values out of $\mathcal{Q}$ to all nodes of the circuit, $s : \mathcal{N} \rightarrow \mathcal{Q}$. Given two states s_1 and 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 in which all nodes are assigned with values out of $\{0, 1\}$ is a ***concrete state***. 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. The notation $\sigma(i), i \in \mathbb{N}$, denotes the state at time i in σ. The notation $\sigma(i)(n), i \in \mathbb{N}, n \in \mathcal{N}$, denotes the value of the node n in the state $\sigma(i)$. The notation $\sigma^i, i \in \mathbb{N}$, 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 with respect to $\sqsubseteq$.

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 $V \cup \mathcal{Q}$. The notions of states and sequences can be extended to the symbolic domain. 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. Given a symbolic state s and an assignment ϕ to V, $\phi(s)$ denotes the state that is obtained by applying ϕ to all symbolic expressions in s.

A ***symbolic sequence*** over V is a series of symbolic states that represents a set of sequences, one for each assignment to the variables in V. Given a symbolic sequence σ and an assignment ϕ to V, $\phi(\sigma)$ denotes the sequence that is obtained by applying ϕ to all symbolic expressions in σ. Given two symbolic sequences σ_1 and σ_2 over V, we say that $\sigma_1 \sqsubseteq \sigma_2$ if for all assignments ϕ to the variables in V, $\phi(\sigma_1) \sqsubseteq \phi(\sigma_2)$.

Sequences may be incompatible with the behavior of M. A ***(symbolic) tra-***

jectory π is a (symbolic) sequence that is compatible with the behavior of M: let $val(n, t, \pi)$ 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), $\pi(t)(n) \sqsubseteq val(n, t, \pi)$ (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 $\pi_c \sqsubseteq \pi$.

2.8 Symbolic Trajectory Evaluation

We now describe the Trajectory Evaluation Language (TEL) used to specify properties for STE.
A ***Trajectory Evaluation Logic*** (TEL) formula is defined recursively over V as follows:

1. **Simple Predicates:** (n is p), where $n \in \mathcal{N}$ and p is a Boolean expression over V.
2. **Conjunction:** ($f_1 \wedge f_2$), where f_1 and f_2 are TEL formulas.
3. **Domain Restriction:** ($p \rightarrow f$), where f is a TEL formula and p is a Boolean expression over V.
4. **Next Time:** ($\mathbf{N}f$), where f is a TEL formula 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. $\mathbf{N^t}$ denotes the application of t next time operators. The constraints on (n, t) are those appearing in the scope of $\mathbf{N^t}$. A TEL formula f has a ***maximal depth***, denoted depth(f), which is the maximal time t for which there exists a constraint in f on some node (n, t), plus one.

Usually, the satisfaction of a TEL formula f on a symbolic sequence σ is defined in the two-valued truth domain [20], i.e., f is either satisfied or not satisfied. In [12], $\mathcal{Q}$ is used also as a four-valued truth domain for an extension of TEL. We also use a four-valued semantics. However, our semantic definition is different from [12] with respect to the $\perp$ value. In [12], 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 with respect to contradictory information about the state space. The intuition behind our definition is that a sequence that contains a $\perp$ 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:
$[\phi, \sigma \models f] = \bot \ \leftrightarrow \ \exists i \geq 0, n \in \mathcal{N} : \phi(\sigma)(i)(n) = \bot$. 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 $\phi(\sigma)(0)(n) \in \{0, 1\}$
$[\phi, \sigma \models n \text{ is } p] = X \ \leftrightarrow \ \phi(\sigma)(0)(n) = X$
$(\phi, \sigma \models p \rightarrow f) = (\neg\phi(p) \vee \phi, \sigma \models f)$
$(\phi, \sigma \models f_1 \wedge f_2) = (\phi, \sigma \models f_1 \wedge \phi, \sigma \models f_2)$
$(\phi, \sigma \models \mathbf{N}f) = \phi, \sigma^1 \models f$

Note that given an assignment ϕ to V, $\phi(p)$ is a constant (0 or 1). In addition, the $\bot$ truth value is determined only according to ϕ and σ, regardless of f. 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 $\exists\phi : [\phi, \sigma \models f] = 1$
$[\sigma \models f] = \bot \ \leftrightarrow \ \forall\phi : [\phi, \sigma \models f] = \bot$

Note that sequences $\phi(\sigma)$ for which the truth value of f is $\bot$ are ignored, since they do not correspond to real executions of the circuit. Only if the truth value of f is $\bot$ for all possible sequences $\phi(\sigma)$, then the truth value of $[\sigma \models f]$ is $\bot$.

Theorem 1 *Given a set of variables V, a TEL formula f over V, symbolic sequences σ_1, σ_2 over V, and an assignment ϕ to V, if $\phi(\sigma_2) \sqsubseteq \phi(\sigma_1)$ then $[\phi, \sigma_2 \models f] \sqsubseteq [\phi, \sigma_1 \models f]$.*

Proof:

We distinguish between two cases: If $\phi(\sigma_2)$ contains a $\bot$ value, then by definition $\phi(\sigma_2) \sqsubseteq \phi(\sigma_1)$ and $[\phi, \sigma_2 \models f] = \bot$ and therefore $[\phi, \sigma_2 \models f] \sqsubseteq [\phi, \sigma_1 \models f]$. Otherwise, the proof is by induction on the structure of f. Note that since $\phi(\sigma_2)$ does not contain $\bot$ and $\phi(\sigma_2) \sqsubseteq \phi(\sigma_1)$, then $\phi(\sigma_1)$ also does not contain a $\bot$ value.

- Induction Basis: $f = (n \text{ is } p)$.

  Since $\phi(\sigma_1)$ does not contain $\bot$, then $[\phi, \sigma_1 \models (n \text{ is } p)] \in \{1, 0, X\}$. Since we assume that $\phi(\sigma_2) \sqsubseteq \phi(\sigma_1)$ and $\phi(\sigma_2)$ does not contain $\bot$, we get that $\phi(\sigma_2)(0)(n) \sqsubseteq \phi(\sigma_1)(0)(n)$. If $\phi(\sigma_1)(0)(n) \in \{0, 1\}$ then $\phi(\sigma_2)(0)(n) = \phi(\sigma_1)(0)(n)$ and therefore $[\phi, \sigma_2 \models (n \text{ is } p)] = [\phi, \sigma_1 \models (n \text{ is } p)]$. Otherwise, $\phi(\sigma_1)(0)(n) = X$, and therefore $[\phi, \sigma_1 \models (n \text{ is } p)] = X$, and we conclude that $[\phi, \sigma_2 \models (n \text{ is } p)] \sqsubseteq [\phi, \sigma_1 \models (n \text{ is } p)]$.

- Induction Step:

1. $f = f_1 \wedge f_2$. By the induction hypothesis, $[\phi, \sigma_2 \models f_1] \sqsubseteq [\phi, \sigma_1 \models f_1]$ and $[\phi, \sigma_2 \models f_2] \sqsubseteq [\phi, \sigma_1 \models f_2]$. Since the $\wedge$ operator is monotonic, we get that $[\phi, \sigma_2 \models f_1 \wedge f_2] \sqsubseteq [\phi, \sigma_1 \models f_1 \wedge f_2]$.
2. $f = p \rightarrow f_1$. If $\phi(p) = 0$, then by definition $[\phi, \sigma_1 \models p \rightarrow f_1] = [\phi, \sigma_2 \models p \rightarrow f_1] = 1$. Otherwise, $\phi(p) = 1$, and by the induction hypothesis, $[\phi, \sigma_2 \models f_1] \sqsubseteq [\phi, \sigma_1 \models f_1]$. Thus, $[\phi, \sigma_2 \models p \rightarrow f_1] \sqsubseteq [\phi, \sigma_1 \models p \rightarrow f_1]$
3. $f = \mathbf{N} f_1$. Since $\phi(\sigma_2) \sqsubseteq \phi(\sigma_1)$, then $\phi({\sigma_2}^1) \sqsubseteq \phi({\sigma_1}^1)$. By the induction hypothesis, $[\phi, {\sigma_2}^1 \models f_1] \sqsubseteq [\phi, {\sigma_1}^1 \models f_1]$. Thus, $[\phi, \sigma_2 \models \mathbf{N} f_1] \sqsubseteq [\phi, \sigma_1 \models \mathbf{N} f_1]$. □

Corollary 1 *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 is proven in [12] 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, $[\sigma \models f] \in \{1, \perp\}$ if and only if $\sigma \sqsubseteq \sigma^f$. For example, $\sigma^{q \rightarrow (n \text{ is } p)}$ is the sequence $s_{(n,q \rightarrow p)} s_x s_x s_x \ldots$, 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.

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, $[\pi \models f] \in \{1, \perp\}$ if and only if $\pi \sqsubseteq \pi^f$. If $[\pi^f \models f] = \perp$ then there is no trajectory π of M and assignment ϕ to V so that $[\phi, \pi \models f] = 1$.

Similar definitions for σ^f and π^f exist in [20] with respect to a two-valued truth domain $\{T, F\}$. T stands for either 1 or $\perp$, and F stands for either 0 or X.

Given σ^f, π^f is computed iteratively as follows: For all i, $\pi^f(i)$ is initialized to $\sigma^f(i)$, and then the value of each node (n, i) is calculated according to the values of its source nodes, and 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 nodes values at time i will be derived. An example of a computation of π^f is given in Example 1.

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 for all 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 $[\pi^A \models C] \in \{1, \perp\}$ then it holds that $M \models (A \Longrightarrow C)$. If $[\pi^A \models C] = 0$

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

Table 2.7: The Defining Sequence σ^A

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

Table 2.8: The Defining Trajectory π^A

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. In this thesis we take the approach that supports the full semantics of STE as defined above, i.e., including the case in which there are occurrences of $\perp$ in π^A. Note that although π^A is infinite, it is sufficient to examine only a bounded prefix of length depth(A) in order to detect $\perp$ values in π^A. The first $\perp$ 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 $\forall i >$ depth$(A) : \sigma^A(i) = s_x$, it must hold that $t \leq$ 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 exist $\phi, i \geq 0, n \in \mathcal{N}$ so that $\phi(\pi^A)(i)(n) \not\sqsubseteq \phi(\sigma^C)(i)(n)$, $\phi(\pi^A)(i)(n) \not\sqsupseteq \phi(\sigma^C)(i)(n)$ and for all $i' \geq 0, n' \in \mathcal{N}$, $\phi(\pi^A)(i')(n') \neq \perp$, then $[\pi^A \models C] = 0$.

3. Otherwise, if there exist $\phi, i \geq 0, n \in \mathcal{N}$ so that $\phi(\pi^A)(i)(n) \sqsupseteq \phi(\sigma^C)(i)(n)$, $\phi(\pi^A)(i)(n) \neq \phi(\sigma^C)(i)(n)$ and for all $i' \geq 0, n' \in \mathcal{N}$, $\phi(\pi^A)(i')(n') \neq \perp$, then $[\pi^A \models C] = X$.

4. Otherwise, $[\pi^A \models C] = 1$.

Note that although π^A and σ^C are infinite, it is sufficient 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 2.1. 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). Table 2.7 describes the defining sequence σ^A of M and A, up to time 1. Table 2.8*

describes the defining trajectory π^A of M and A, up to time 1. Both tables contain the symbolic expression of each node at time 0 and 1. The states $\sigma^A(i)$ and $\pi^A(i)$ are 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 are assigned X at all times, except for node N6 at time 1, which is assigned 1. $[\pi^A \models C] = 0$ due to the assignments to V in which $v_1 = 0$. We will return to this example in Section 5.

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 the node (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.

Chapter 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 expected 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 chapter 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 first describe the handling of $\perp$ values which is required for the description of the general abstraction and refinement 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 sufficient 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 $\phi(\pi^A)(t')(n') = \perp$. Thus, $[\pi^A \models C] = \perp$ if and only if $nbot \equiv 0$.

We now describe how the abstraction and refinement 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: $(\bigvee_{(n,t)\in C}((g^1_{n,t} \wedge \neg f^1_{n,t} \wedge f^0_{n,t}) \vee (g^0_{n,t} \wedge f^1_{n,t} \wedge \neg f^0_{n,t}))) \wedge nbot$. Each assignment ϕ in

this set represents a counterexample $\phi(\pi^A)$. 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: $(\bigvee_{(n,t)\in C}((g^1_{n,t} \vee g^0_{n,t}) \wedge \neg f^1_{n,t} \wedge \neg f^0_{n,t})) \wedge nbot$. 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 correct. 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.

In order to preserve the semantics of $A \Longrightarrow C$, we require that $M \models A_{new} \Longrightarrow C$ if and only if $M \models A \Longrightarrow C$.

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 inputs 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 4.2, a small but significant addition is made to our refinement technique.

3.1 The Relationship Between the Abstract and Refined Assertions

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. We define A_{new} to be a refinement of A over $V \cup V_{new}$, where A_{new} is obtained 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: $A_{new} = A \wedge \bigwedge_{(in,t)\in PI_{ref}} \mathbf{N}^t(p \rightarrow (in \text{ is } v_{in,t}))$, where $p = \neg q$ if (in,t) has a constraint $\mathbf{N}^t(q \rightarrow (in \text{ is } e))$ 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 $q \rightarrow f$. For example, if A_{org} contains a constraint $(v_1 \rightarrow n \text{ is } 1)$ and $(n,0)$ is chosen for refinement, then the added constraint is $(\neg v_1 \rightarrow n \text{ is } v_{n,0})$

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 $[(\phi \cup \phi'), \pi \models A_{new}] = 1$. Note that although π is concrete, an assignment to $V \cup V_{new}$ is still required, since A_{new} is defined over $V \cup V_{new}$.

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

Theorem 2 implies that the set of all concrete trajectories of M on which A_{new} holds for some assignment ϕ_1 to $V \cup V_{new}$ and the set of all concrete trajectories of M on which A holds for some assignment ϕ_2 to V are identical.

Proof: Let π be a concrete trajectory so that $\pi \in runs(A, M, \phi)$. Thus, $[\phi, \pi \models A] = 1$. Let $\pi(t)(in)$ be the value that each input $(in,t) \in PI_{ref}$ receives in the trajectory π. Let ϕ' be an assignment that gives for each variable $v_{in,t} \in V_{new}$ that is attached to the input $(in,t) \in PI_{ref}$ the value $\pi(t)(in)$. Since each variable $v_{in,t} \in V_{new}$ is attached to a single input $(in,t) \in PI_{ref}$ and appears only in the new condition $\mathbf{N}^t(p \rightarrow (in \text{ is } v_{in,t}))$, then $[\phi \cup \phi', \pi \models A_{new}] = 1$. Thus, $\pi \in runs(A_{new}, M, \phi)$.

Let π be a concrete trajectory so that $\pi \in runs(A_{new}, M, \phi)$. Thus, there exists an assignment ϕ' to V_{new} so that $[\phi \cup \phi', \pi \models A_{new}] = 1$. Since the set of

conditions on nodes in A is included in the set of conditions on nodes in A_{new}, this means that $[\phi \cup \phi', \pi \models A] = 1$. Since A is over the variables in V only, we get that $[\phi, \pi \models A] = 1$. Thus, $\pi \in runs(A, M, \phi)$. ☐

Theorem 3 *If* $[\pi^{A_{new}} \models C] = 1$ *then for all* ϕ *it holds that* $\forall \pi \in runs(A, M, \phi)$: $[\phi, \pi \models C] = 1$.

Theorem 3 implies that if $A_{new} \Longrightarrow C$ holds on all concrete trajectories of M, then so does $A \Longrightarrow C$.

Proof: $[\pi^{A_{new}} \models C] = 1$ implies that $\forall \phi(V \cup V_{new}) : [\phi, \pi^{A_{new}} \models C] \in \{1, \bot\}$. Since C is over the variables in V only, this means that $\forall \phi(V)\ \forall \phi'(V_{new})$: $[(\phi \cup \phi'), \pi^{A_{new}} \models C] \in \{1, \bot\}$. Due to monotonicity of the satisfaction relation, we get that $\forall \phi(V) \forall \pi \in runs(A_{new}, M, \phi) : [\phi, \pi \models C] \in \{1, \bot\}$. Since all trajectories in $runs(A_{new}, M, \phi)$ are concrete, we get that $\forall \phi(V) \forall \pi \in runs(A_{new}, M, \phi) : [\phi, \pi \models C] = 1$. Since $runs(A, M, \phi) = runs(A_{new}, M, \phi)$, we conclude that $\forall \phi(V) \forall \pi \in runs(A, M, \phi) : [\phi, \pi \models C] = 1$. ☐

Theorem 4 *If there exists* 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 implies that if $A_{new} \Longrightarrow C$ yields a concrete trajectory π which constitutes a counterexample, then π is also a concrete counterexample with respect to $A \Longrightarrow C$.

Proof: By definition, $\pi \in runs(A_{new}, M, \phi)$. According to Theorem 2, $\pi \in runs(A, M, \phi)$. Thus, $[\phi, \pi \models A] = 1$. Since C is defined over V only and $[\phi \cup \phi', \pi \models C] = 0$, we get that $[\phi, \pi \models C] = 0$. ☐

We would also like to define the connection between abstract (non concrete) trajectories before and after refinement:

Theorem 5 $\forall \phi(V) \forall \phi'(V_{new}) : (\phi \cup \phi')(\pi^{A_{new}}) \sqsubseteq \phi(\pi^A)$.

Proof: We will prove that $\forall t \forall n \in \mathcal{N}\ \forall \phi(V) \forall \phi'(V_{new}) : (\phi \cup \phi')(\pi^{A_{new}})(t)(n) \sqsubseteq \phi(\pi^A)(t)(n)$ by induction on the structure of the defining trajectory:

- Induction basis: The defining trajectory at time 0.

  We will prove the induction basis by induction on the structure of M. We denote by $d(n, 0)$ the maximal distance in backward edges of a node $(n, 0)$ from an input or a latch. Since combinational loops are not allowed, $d(n, 0)$ is well-defined. We will prove the induction basis for all nodes $(n, 0)$ for which $d(n, 0) = k$.

– Induction basis: $d(n, 0) = 0$.

  The nodes for which $d(n, 0) = 0$ are exactly the inputs and latches at time 0. Due to the construction of A_{new} from A, it holds that for all inputs and latches $(n, 0)$, $\forall\phi(V)\forall\phi'(V_{new}) : (\phi \cup \phi')(\pi^{A_{new}})(0)(n) \sqsubseteq \phi(\pi^A)(0)(n)$. This is due to the fact that for all $\phi(V)$, if A constrains $(n, 0)$ to 0 or 1 then A_{new} constrains it to the same value, and if A does not constrain it, then $\phi(\pi^A)(0)(n) = X$.

– Induction step: We assume that for all nodes $(n, 0)$ with $d(n, 0) < k$ it holds that $\forall\phi(V)\forall\phi'(V_{new}) : (\phi \cup \phi')(\pi^{A_{new}})(0)(n) \sqsubseteq \phi(\pi^A)(0)(n)$.

  Let $(n, 0)$ be a node with $d(n, 0) = k$. For all source nodes $(m, 0)$ of $(n, 0)$ it holds that $d(m, 0) < k$, otherwise it would hold that $d(n, 0) > k$ in contradiction to our assumption. Thus, the induction hypothesis holds for the source nodes of $(n, 0)$. Given a node (n, t) and a trajectory π, let $val(n, t, \pi)$ be the value of (n, t) as computed according to its source nodes values in π. Due to the construction of A_{new} from A, A and A_{new} contain exactly the same constraints on combinational nodes. For each combinational node $(n, 0)$ and assignments $\phi(V)$,$\phi'(V_{new})$, if A (and A_{new}) does not constrain $(n, 0)$, then $\phi(\pi^A)(0)(n) = val(n, 0, \phi(\pi^A))$ and $(\phi \cup \phi')(\pi^A_{new})(0)(n) = val(n, 0, (\phi \cup \phi')(\pi^A_{new}))$. Since all Boolean operators are monotonic with respect to Q, it holds that $val(n, 0, (\phi \cup \phi')(\pi^A_{new})) \sqsubseteq val(n, 0, \phi(\pi^A))$, and thus $(\phi \cup \phi')(\pi^{A_{new}})(0)(n) \sqsubseteq \phi(\pi^A)(0)(n)$. Otherwise, A and A_{new} constrain $(n, 0)$ to the same value $b \in \{0, 1\}$. It holds that $\phi(\pi^A)(0)(n) = val(n, 0, \phi(\pi^A)) \sqcap b$ and $(\phi \cup \phi')(\pi^A_{new})(0)(n) = val(n, 0, (\phi\cup\phi')(\pi^A_{new}))\sqcap b$. Since $val(n, 0, (\phi\cup\phi')(\pi^A_{new})) \sqsubseteq val(n, 0, \phi(\pi^A))$ and $\sqcap$ is monotonic with respect to Q, we get that $(\phi \cup \phi')(\pi^{A_{new}})(0)(n) \sqsubseteq \phi(\pi^A)(0)(n)$.

• Induction step: We assume that $\forall 0 \leq t' < t \forall n \in \mathcal{N} \forall\phi(V)\forall\phi'(V_{new}) : (\phi \cup \phi')(\pi^{A_{new}})(t')(n) \sqsubseteq \phi(\pi^A)(t')(n)$.

  We again prove the induction step by induction on the structure of M. For all $t > 0$, we denote by $d(n, t)$ the maximal distance in backward edges of a node (n, t) from an input at time t or from a node at time $t - 1$.

  – Induction basis: $d(n, t) = 0$.

    The induction basis is proven in the same way as the previous induction basis, using $d(n, t)$ instead of $d(n, 0)$. The only difference is that the set of nodes $(n, t), t > 0$ for which $d(n, t) = 0$ includes only the inputs at time t.

– Induction step: $d(n, t) = k$. The induction step is proven in the same way as the previous induction step, using $d(n, t)$ instead of $d(n, 0)$. The only difference is that the source nodes of a node $(n, t), t > 0$ may also include nodes from time $t - 1$. The external induction hypothesis holds on these nodes, and thus the induction hypothesis holds on all source nodes of (n, t), and the proof of the previous induction step can be applied here as well. □

Chapter 4

Selecting Inputs for Refinement

After choosing our refinement methodology, we need to describe how exactly the refinement process is performed. In this section we assume that $[\pi^A \models C] = X$. 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 automatic 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.

4.1 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 $[\phi, \pi^A \models C] = X$. Each such trajectory $\phi(\pi^A)$ is called an ***incomplete trajectory***. This trace may contain multiple nodes that are required by C to have a definite value (either 0 or 1) but equal X. We refer to such nodes as ***undecided nodes***. In automatic refinement we need to decide whether to eliminate all incomplete trajectories at once or one at each refinement iteration, and whether to eliminate all undecided nodes at once or one at each refinement iteration. 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 undesired X value that is eliminated may be replaced in the next iteration with a definite value that contradicts the required value (a counterexample). In such a case, the current abstraction and refinement loop is terminated without all undecided nodes being eliminated. We suggest to choose an undecided node (n, t) with a minimal number of inputs in its BCOI, since it is likely that such an undecided node will require the addition of less constraints than an undecided node that depends on more inputs.

Algorithm 1 EliminateIrrelevantPIs((n, t))

sinks_relevant $\leftarrow \bigvee_{(m,t') \in out(n,t)}$ relevant$_{m,t'}$
relevant$_{n,t} \leftarrow$ sinks_relevant $\wedge \neg f^0_{n,t} \wedge \neg f^1_{n,t}$

Out of those, we choose an undecided node with a minimal number of nodes in its BCOI, with the motivation to minimize the additional computation effort that will be required after the addition of the new constraints. Our experimental results support this choice. The chosen undecided node is our ***refinement goal*** and is denoted $(root, tt)$.

We also decide 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 symbolic 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 assignments ϕ for which $(root, tt)$ is undecided in the incomplete trajectory $\phi(\pi^A)$. 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.

4.2 Eliminating irrelevant inputs

Once we have a refinement goal $(root, tt)$, we need to choose input nodes (in, t) for which constraints will be added to A. Naturally, only inputs in the BCOI of $(root, tt)$ should be considered. However, out of these inputs, some 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 the execution of M over time, 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. If no such ϕ exists, then (in, t) is ***irrelevant*** to $(root, tt)$.

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: $(\neg f^1_{root,tt} \wedge \neg f^0_{root,tt} \wedge (g^1_{root,tt} \vee g^0_{root,tt})) \wedge nbot$. 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 do not satisfy relevant$_{in,t}$, changing (in, t) from X to a definite

value in $\phi(\pi^A)$ cannot 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. This is done using the domain restriction operation: 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 another refinement goal $(root', tt')$, then the previous domain restriction is extended by disjunction to include the condition under which (in, t) is relevant to $(root', tt')$. Theorems 2, 3 and 4 from Chapter 3 hold also for the optimized refinement. Let PI be the set of inputs of the circuit. Then the set of all inputs that are relevant to $(root, tt)$ is $PI_{(root,tt)} \equiv \{(in, t) \mid in \in PI \land \text{relevant}_{in,t} \not\equiv 0\}$. 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 the elimination of all undesired X values from $(root, tt)$. Namely, adding constraints for all inputs in $PI_{(root,tt)}$ will guarantee elimination of all cases in which $(root, tt)$ is undecided. However, it is possible that adding constraints for only a subset of $PI_{(root,tt)}$ will 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 with respect to $(root, tt)$.

4.3 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 increasing 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. A motivation for this is that a 1 or 0 truth value may be reached even without adding constraints for all relevant inputs.

A subset of $PI_{(root,tt)}$ is heuristically selected for refinement as described in Algorithm 2. Each node (n, t) selects a subset of $PI_{(root,tt)}$ as candidates for refinement, held in candidates$_{n,t}$. The final set of inputs for refinement is selected out of 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 out(n, t) return the sink nodes of (n, t) that are in the BCOI of $(root, tt)$, and let 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), 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 exist 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 source nodes, 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. It may have other control source nodes such as set and reset. 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$_{n,t}$.

Algorithm 2 SelectBestPIs($(root, tt), PI_{(root,tt)}$)

---

for all $(in, t) \in PI$ **do**
  if $(in, t) \in PI_{(root,tt)}$ **then**
    candidates$_{in,t} \leftarrow \{(in, t)\}$
  else
    candidates$_{in,t} \leftarrow \emptyset$
  end if
  for all $(n, t') \in$ out(in, t) **do**
    count$_{n,t'}$ $++$
    if degin$(n, t') =$ count$_{n,t'}$ **then**
      SelectBestPIsRec((n, t'))
    end if
  end for
end for

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

Algorithm 3 SelectBestPIsRec((n, t))

```
if isEmpty(sourceCand_{n,t}) then
    candidates_{n,t} ← ∅
else if existMutualCanidates(sourceCand_{n,t}) then
    candidates_{n,t} ← mutualCanidates(sourceCand_{n,t})
else if hasControlSourceCandidates((n, t)) then
    candidates_{n,t} ← controlSourcesCandidates((n, t))
else
    candidates_{n,t} ← majorityCandidates((n, t))
end if
for all (m, t') ∈ out(n, t) do
    count_{m,t'} + +
    if degin(m, t') = count_{m,t'} then
        SelectBestPIsRec((m, t'))
    end if
end for
```

of a data input is required for verification, it can be constrained 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. We restrict the number of inputs (in, t) in the final set of candidates to a strict number l. If the number of candidates of $(root, tt)$ exceeds l, we select the ones with a minimal number of nodes in their output cone, in order to minimize the additional computation effort that will be required after the addition of the new constraints. Out of them, we randomly choose the inputs for refinement.

Chapter 5

Detecting Vacuity and Spurious Counterexamples

In this chapter we raise the problem of hidden vacuity and spurious counterexamples that may occur due to the abstraction 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 expected 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 chapter 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 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 (or an initial value of a latch), 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 [18]. This is because the constraints imposed by A are also incorporated by applying the $\sqcap$ operator on trajectories that may be abstract, only that they are not necessarily the defining trajectory of A.

Example 2 *We return to Example 1 from Chapter 2. Note that the defining trajectory π^A does not contain $\perp$ values. In addition, $[\pi^A \models C] = 0$ due to all 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 assignment ϕ to V, C under ϕ may impose no requirements. This is due to the Domain Restriction operation, where the constraint is of the form $p \rightarrow f$ and $\phi(p)$ 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 [4] for ACTL.

We say that $A \Longrightarrow C$ ***passes vacuously*** on M if $A \Longrightarrow C$ is vacuous in M and $[\pi^A \models C] \in \{\perp, 1\}$. Note that if $[\pi^A \models C] = \perp$, then surely $A \Longrightarrow C$ passes vacuously, and thus vacuity detection is not required in this case.

A counterexample π is ***spurious*** if there is no concrete trajectory π_c of M so that $\pi_c \sqsubseteq \pi$. Given the defining trajectory π^A, we say that the symbolic counterexample ce is ***spurious*** if for all assignments ϕ to V that satisfy 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 the symbolic counterexample ce is spurious.

For implementations that do not compute π^A such as [18], we say that $A \Longrightarrow C$ fails vacuously on M if the result of STE is fail and the produced counterexample is spurious.

As explained before, vacuity detection is required only when A constrains internal nodes. 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, π should also constitute a counterexample for $A \Longrightarrow C$. We propose two different algorithms for vacuity detection. The first algorithm uses Bounded Model Checking (BMC) [5] 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.

An additional vacuity problem that arises in constraint-based STE [18] is described in Appendix A.

5.1 Vacuity Detection using Bounded Model Checking

Since A can be expressed as an LTL formula, we can translate A and M into a 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 $A \implies C$ passed non-vacuously. Otherwise, we conclude that $A \implies C$ 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 $\bot$, since the BMC formula runs on the concrete model, and thus the domain of the nodes in the BMC formula is Boolean. For other implementations such as [25, 18], ce consists of an assignment of values to V and to the circuit nodes at different times. If BMC finds a satisfying assignment to the resulting formula, the assignment of BMC to the nodes in M constitutes a concrete counterexample for $A \implies C$. Otherwise, we conclude that $A \implies C$ 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. It also contains an additional constraint in the form of a Boolean expression which by itself does not constrain 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.

5.2 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 the constraints of A on inputs, and A^{out} is a TEL formula that contains exactly 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 Chapter 3, 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 4 describes vacuity detection using STE. It received 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 4 STEVacuityDetection(A, C, ce)

1: $A^{in} \leftarrow$ inputConstraints(A)
2: $A^{out} \leftarrow$ internalConstraints(A)
3: $\Phi \leftarrow \bigwedge_{(n,t)\in A^{out}}((g^1_{n,t} \wedge f^1_{n,t}) \vee (g^0_{n,t} \wedge f^0_{n,t}))$
$\{\Phi$ 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 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 $[\phi, \pi^{A^{in}} \models A^{in}] = 1$ by definition of the defining trajectory, 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 the constraints in A^{in} and A^{out}. According to Theorem 5, we get that $\phi(\pi^{A^{in}}) \sqsubseteq \phi(\pi^{A})$. 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 4 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 - the 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 4 is that it exploits the abstraction in STE, whereas vacuity detection using BMC runs on the concrete model.

5.2.1 Vacuity Detection using satGSTE

As an alternative to Algorithm 4, we present a vacuity detection algorithm that uses SAT-based STE. The motivation for using SAT-based STE is that only one witness or counterexample is produced. satGSTE [25] is a SAT-based STE implementation in which the first stage is to compute the defining trajectory using semi-canonical data structures named bexpr that represent Boolean expressions. This stage guarantees that the size of the problem depends only on the size of A. In the second stage, the expressions of the nodes appearing in C are translated into a SAT formula, and a constraint is added that for at least one of the nodes appearing in C, its value in the defining trajectory does not correspond to the value imposed by its requirement in C. If there is no satisfying assignment to the SAT formula, then $A \Longrightarrow C$ holds on M. Otherwise, the satisfying assignment is analyzed to determine whether it constitutes a counterexample or an incomplete trace. In the latter case, refinement of $A \Longrightarrow C$ is required.

satGSTE can also be used for vacuity detection using STE in the following way: satGSTE is run on the STE assertion $A^{in} \Longrightarrow A^{out}$. After computing the defining trajectory of A^{in} using bexpr, a SAT formula is produced in which the values of the nodes in the defining trajectory are constrained to be *equal* to the values imposed by the requirements in A^{out}. Additional constraints are added to the SAT formula as done in vacuity detection using BMC in order to guarantee that the satisfying assignment imposes some requirement in C in case $[\pi^{A} \models C] = 1$, and

that the satisfying assignment constitutes a counterexample in case $[\pi^A \models C] = 0$. If a satisfying assignment is found, then we conclude that the result of STE on $A \Longrightarrow C$ is non-vacuous. Otherwise, we need to check whether there exists an assignment ϕ to V so that $[\phi, \pi^{A^{in}} \models A^{out}] = X$. This is done by changing the SAT formula so that the values of the nodes that appear in A^{out} are now constrained to be *greater than or equal* to the values imposed by the requirements in A^{out}, with respect to the partial order $\sqsubseteq$. If a satisfying assignment is found, then refinement of A^{in} is required in order to detect (non)-vacuity. Otherwise, we conclude that the result of STE on $A \Longrightarrow C$ is vacuous.

5.3 Preliminary Stage in 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 Chapter 2, 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.

Chapter 6

Experimental Results

We implemented our automatic refinement algorithm on top of STE in the Intel FORTE environment [21]. Our **AutoSTE** algorithm 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 in C. The chosen node has minimal time and minimal number of inputs and nodes in its BCOI. Next, Algorithm 1 from Section 4.2 computes the set of relevant inputs (in, t). Heuristics are applied in order to choose a subset of those inputs, as described in Section 4.3. In our experimental results we restrict the number of refined candidates in each iteration to 1. We change A as described in Section 4.2 and STE is rerun on the new assertion.

We ran our algorithm on two different circuits. All runs were performed on a 3.2 GHz Pentium 4 computer with 4 GB memory. The first circuit is the Content Addressable Memory (CAM) from Intel's GSTE tutorial. The second circuit is IBM's Calculator 2 design [23]. It has a complex specification and is challenging for Model Checking. Therefore, it constitutes a good example for the benefit the user can gain from automatic refinement in STE.

We ran Forte in default mode which considers antecedent failures as illegal. However, since all of our assertions do not contain constraints on internal nodes, no antecedent failures can occur.

6.1 Content Addressable Memory

The CAM shown in Figure 6.1 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).

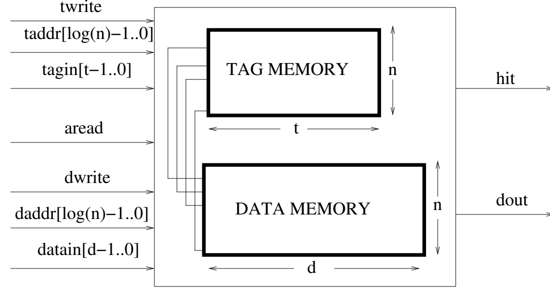

Figure 6.1: Content Addressable Memory. Tag size=t, Number of entries=n, Data size=d

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

Table 6.1: Automatic Refinement Performance on CAM Assertions

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 [17]. The assertions in [17] 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 when bounding the time that passed between the time the tag is written and the time it is read. We present the results of **AutoSTE** on 3 such assertions. Figure 6.1 reports the final result, number of refinement iterations, run-time in seconds and peak BDD nodes for each assertion. Table 6.2 reports the refinement goal and added constraint in each refinement iteration. $v_{n,t}$ denotes a fresh symbolic variable for node (n, t). Similarly, $\overrightarrow{v}_{n,t}$ is a vector of fresh symbolic variables for the vector of nodes n at time t.
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. Assertion 1 is: (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. The reason is that 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 $\overrightarrow{A}$ [1]. If there is a write operation to the tag memory at time 1 (twrite is 1), then $\overrightarrow{\text{TAG}}$ should be found since it is written again to the tag memory. However, $[\pi^A \models C] = X$.

[1]usually it is assumed that the same tag cannot appear in more than one tag memory entry. However, even if we do not make this assumption, assertion 1 should still pass.

The reason is that since twrite and taddr at time 1 equal X, the CAM cannot determine whether to write the value of tagin at time 1 to the tag memory. Moreover, it cannot decide to which tag entry to write it. As a result, the value of the entire tag memory at time 1 is X, and thus the value of hit at time 1 is X.

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

Table 6.2: Automatic Refinement of CAM Assertions

After two refinements, **AutoSTE** terminates with a pass result. Note that constraints were added only in the subset of trajectories in which they were necessary for verification. $\overrightarrow{\text{TAG}} \neq 0$ appears in the domain restriction since in this specific CAM implementation, the default value of the data source nodes of the tag memory is 0. Thus, in the special case where $\overrightarrow{\text{TAG}} = 0$, even without knowing if and to which tag entry a tag is written at time 1, the CAM can determine that an entry holding 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 $\overrightarrow{D}$. **AutoSTE** produces a counterexample for assertion 2. In the first two iterations the refinement goal was (hit,1), since it depends on less inputs than each entry of dout. Once the requirement on $(hit, 1)$

holds, the next refinement goal is dout[0]. In the following two iterations, twrite and taddr at time 1 are added to A when $\overrightarrow{\text{TAG}} = 0$, since in order to determine the value of dout[0] at time 1, it should be known which tag entry actually holds the tag that is written at time 1 (if written at all). The set of relevant PIs for refinement in iterations 5-7 included dwrite, daddr and din[0], all at times 0 and 1, the initial values of all tag memory entries, and the initial values of bit number 0 of all data memory entries.

The final refined assertion yields a symbolic 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 the fact that the assertion is erroneous. 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 to the assertion.

Assertion 3 states that if at time 0 $\overrightarrow{\text{TAG}}$ is written to address $\overrightarrow{A}$ and datamem[$\overrightarrow{A}$] is $\overrightarrow{D}$, and at time 2 $\overrightarrow{\text{TAG}}$ is read, and in addition no write was performed to the tag and data memory since time 0, then at time 2 hit is 1 and dout is $\overrightarrow{D}$. 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.

6.2 Calculator Design

The Calculator 2 design [23] shown in Figure 6.2 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 (shiftr) and shift left (shiftl). *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 following cycle. Up to 4 commands can

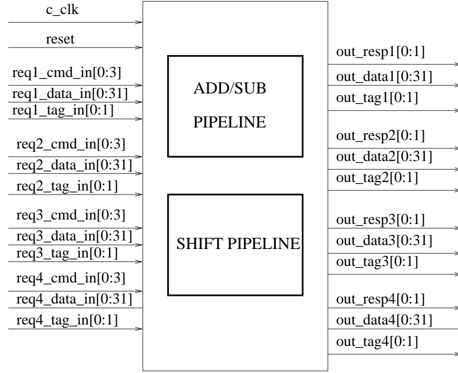

Figure 6.2: Calculator

be sent into the calculator from each of the 4 ports. The tag is a unique identifer for each of the commands from each port. 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 (add/sub and shift) 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.

We present the results of **AutoSTE** on 4 assertions, all resulting in an X truth value of C in the initial STE run. One assertion passed and the others failed. Two assertions failed due to an erroneous specification, and one due to a bug that was planted into the design and is documented in [23]. Figure 6.3 reports the final result, number of refinement iterations, run-time in seconds and peak BDD nodes for each assertion. Table 6.4 reports the refinement goal and added constraint in each refinement iteration.

Assertion 1 checks whether after reset, if a port sends either an add or sub command, and the other ports send either no command or a command different than add and sub, then the port that sent the add/sub command receives a 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. The constraints in which i appears are duplicated

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

Table 6.3: Automatic Refinement Performance on Calculator Assertions

for each constant. We used parametric representation to produce the Boolean expressions $(add \vee sub)$ and $(\neg add \wedge \neg sub)$ over V^2:
$((\overrightarrow{P} = i) \rightarrow (\mathbf{N}^3((\text{req}i\text{_cmd_in is } (add \vee sub)) \wedge (\text{req}i\text{_tag_in is } \overrightarrow{\text{TAG}})$
$\wedge \forall j \neq i : (\text{req}j\text{_cmd_in is } (\neg add \wedge \neg sub))) \wedge \mathbf{N}^4(\text{req}i\text{_cmd_in is none}))) \Longrightarrow$
$((\overrightarrow{P} = i) \rightarrow ((\mathbf{N}^7(\text{out_resp}i \text{ is } 1) \ \wedge (\text{out_tag}i \text{ is } \overrightarrow{\text{TAG}}))))$

The symbolic counterexample presents a case in which there is an overflow for an add command in the data sent by port 1, which triggers an invalid response at cycle 7. Though the cone of influence of out_resp1[0] contains all command, tag and data inputs from all ports at different times, the set of relevant inputs contained only all entries of req1_data_in at cycles 3 and 4. Out of them, the values of req1_data_in[31] at cycles 3 and 4 are the minimal set that should suffice for generating a counterexample, and they are indeed the ones chosen by our heuristics and added as constraints.

In assertion 2 we constrained the command sent by port i to add. The msb bits of the sent data were constrained to 0 to avoid a possibility for overflow. We added a requirement for the output data for port i to match the expected data. However, we removed the constraint on the commands sent by other ports. Assertion 2 is as follows:
$((\overrightarrow{P} = i) \rightarrow (\mathbf{N}^3((\text{req}i\text{_cmd_in is add}) \wedge (\text{req}i\text{_tag_in is } \overrightarrow{\text{TAG}}) \wedge$
$(\text{req}i\text{_data_in[0:30] is } \overrightarrow{A}) \wedge (\text{req}i\text{_data_in[31] is 0})) \wedge \mathbf{N}^4((\text{req}i\text{_cmd_in is none}) \wedge$
$(\text{req}i\text{_data_in[0:30] is } \overrightarrow{B}) \wedge (\text{req}i\text{_data_in[31] is 0})))) \Longrightarrow$
$((\overrightarrow{P} = i) \rightarrow \mathbf{N}^7((\text{out_resp}i \text{ is } 1) \ \wedge (\text{out_tag}i \text{ is } \overrightarrow{\text{TAG}}) \wedge (\text{out_data}i \text{ is } \overrightarrow{A + B})))$

The counterexample displays a case in which both ports 1 and 2 send an add command, and port 3 sends a shift command. In this case port 1 is answered before port 2. The assertion fails due to an erroneous specification: since port 1 has priority over port 2, it is not guaranteed that port 2 will 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 3 presents the following constraints: after reset, a port sends an add or sub command. After 2 cycles, it sends an add command with a certain tag and data arguments, while limiting the msb of the sent data to 0 to avoid a possibility for overflow. Moreover, all other ports do not send an add or sub command throughout this time. The requirements are: the port that sent the add command receives the response good with the appropriate tag value and expected output data. Assertion 3 is as follows:

[2]Constraints on reset and on the clock that exist in all assertions were omitted. Next time operator refers to the calculator clock cycles.

$(\overrightarrow{P} = i) \rightarrow$ ($\mathbf{N}^3$((req$i$_cmd_in is $(add \vee sub)) \wedge \forall j \neq i$: (req$j$_cmd_in is $(\neg add \wedge \neg sub))) \wedge \mathbf{N}^4$(req$j$_cmd_in is none) $\wedge \forall j \neq i$: (req$j$_cmd_in is $(\neg add \wedge \neg sub))) \wedge$ $\mathbf{N}^5$((req$i$_cmd_in is add) $\wedge$ (req$i$_tag_in is $\overrightarrow{TAG}$) $\wedge$ (req$i$_data_in[0:30] is $\overrightarrow{A}$)$\wedge$
(req$i$_data_in[31] is 0)) $\wedge \forall j \neq i$: (req$j$_cmd_in is $(\neg add \wedge \neg sub)$)$\wedge$
$\mathbf{N}^6$((req$i$_cmd_in is none)$\wedge$(req$i$_data_in[0:30] is $\overrightarrow{B}$)$\wedge$(req$i$_data_in[31] is 0)$\wedge$ $\forall j \neq i$: (req$j$_cmd_in is $(\neg add \wedge \neg sub)$))) $\Longrightarrow$
$(\overrightarrow{P} = i) \rightarrow \mathbf{N}^9$((out_resp$i$ is good) $\wedge$(out_tag$i$ is $\overrightarrow{TAG}$)$\wedge$(out_data$i$ is $\overrightarrow{A+B}$))

Note the variables used for the parametric representation of the commands for ports j are different in each cycle.

There was one refinement iteration. The BCOI of resp_out1[0] includes all data and tag inputs of all ports. However, the set of relevant inputs includes only the tags of all ports at cycles 3-5. 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. This counterexample stems from a planted design bug documented in [23]. There is supposed to be no restriction on tag ordering. However, commands whose tags are out of order are classified as an invalid command. When fixing this bug, the assertion passes in the first STE run.

Assertion 4 checks whether after reset, if any port sends an invalid command, then it receives an invalid command response at the first possible cycle. Again we used parametric representation for all combinations of illegal command values (denoted *invalid*):

$(\overrightarrow{P} = i) \rightarrow \mathbf{N}^3$((req$i$_cmd_in is invalid) $\wedge$ (req$i$_tag_in is $\overrightarrow{TAG}$)) $\Longrightarrow$ $(\overrightarrow{P} = i) \rightarrow \mathbf{N}^7$((out_resp$i$ is 2) $\wedge$ (out_tag$i$ is $\overrightarrow{TAG}$))

This assertion should pass. However, due to the implementation of the priority queue, the values of 2 other port commands (and in some cases 3) should be definite in order for the assertion to pass. The set of relevant inputs contained all cmd and data inputs at cycles 3 and 4. The tag inputs were not relevant, although they are all in the BCOI of all output responses. For this assertion, as opposed to others, not using the domain restriction operation reduces the number of refinements.

Assert.	Iter.	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 $\overrightarrow{v}_{req1_tag_in,3}$
4	1	out_resp1[0],7	$\mathbf{N}^3\overrightarrow{P} = 1 \rightarrow$ req2_cmd_in is $\overrightarrow{v}_{req2_cmd_in,3}$
4	2	out_resp1[0],7	$\mathbf{N}^3\overrightarrow{P} = 1 \rightarrow$ req3_cmd_in is $\overrightarrow{v}_{req3_cmd_in,3}$
4	3	out_resp1[0],7	$\mathbf{N}^3(\overrightarrow{P} = 1\wedge$ $(\overrightarrow{v}_{req2_cmd_in,3} = 2 \vee \overrightarrow{v}_{req3_cmd_in,3} = 2)) \rightarrow$ req4_cmd_in is $\overrightarrow{v}_{req4_cmd_in,3}$
4	4	out_resp2[0],7	$\mathbf{N}^3\overrightarrow{P} = 2 \rightarrow$ req1_cmd_in is $\overrightarrow{v}_{req1_cmd_in,3}$
4	5	out_resp2[0],7	$\mathbf{N}^3\overrightarrow{P} = 2 \rightarrow$ req3_cmd_in is $\overrightarrow{v}_{req3_cmd_in,3}$
4	6	out_resp2[0],7	$\mathbf{N}^3(\overrightarrow{P} = 2\wedge$ $(\overrightarrow{v}_{req1_cmd_in,3} = 2 \vee \overrightarrow{v}_{req3_cmd_in,3} = 2)) \rightarrow$ req4_cmd_in is $\overrightarrow{v}_{req4_cmd_in,3}$
4	7	out_resp3[0],7	$\mathbf{N}^3\overrightarrow{P} = 3 \rightarrow$ req1_cmd_in is $\overrightarrow{v}_{req1_cmd_in,3}$
4	8	out_resp3[0],7	$\mathbf{N}^3\overrightarrow{P} = 3 \rightarrow$ req2_cmd_in is $\overrightarrow{v}_{req2_cmd_in,3}$
4	9	out_resp3[0],7	$\mathbf{N}^3(\overrightarrow{P} = 3\wedge$ $(\overrightarrow{v}_{req1_cmd_in,3} = 2 \vee \overrightarrow{v}_{req2_cmd_in,3} = 2)) \rightarrow$ req4_cmd_in is $\overrightarrow{v}_{req4_cmd_in,3}$
4	10	out_resp4[0],7	$\mathbf{N}^3\overrightarrow{P} = 4 \rightarrow$ req1_cmd_in is $\overrightarrow{v}_{req1_cmd_in,3}$
4	11	out_resp4[0],7	$\mathbf{N}^3\overrightarrow{P} = 4 \rightarrow$ req2_cmd_in is $\overrightarrow{v}_{req2_cmd_in,3}$

Table 6.4: Automatic Refinement of Calculator Assertions

Chapter 7

Conclusions and Future Work

This work is a first attempt at automatic refinement of STE assertions. We have developed an automatic refinement technique which is based on heuristics. We proved that 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 explored different aspects of vacuity in STE, including general STE vacuity and vacuity in constrained-based STE, and proposed several methods for vacuity detection.

Future research directions include extending automatic refinement to GSTE [28]. GSTE supports all ω-regular properties and thus requires a more expressive specification language and a reparameterization algorithm. Vacuity definition and detection should also be extended to GSTE.

We would like to implement our suggested vacuity detection algorithms and compare their performance. In addition, we would like to adapt our automatic refinement techniques to SAT based STE [25, 18], and integrate SAT based refinement techniques [16, 7].

Appendix A

Additional Vacuity in Constraint-based STE

In addition to the vacuity problem discussed in Chapter 5, to which we refer as *general STE vacuity*, a special vacuity problem exists in a SAT-based STE implementation named *constraint-based STE* that is suggested in [18]. We refer to this special vacuity problem as *constraint-based STE vacuity*. Although constraint-based STE vacuity is related to general STE vacuity, it does not exist in other STE implementations. In order to understand the causes for constraint-based STE vacuity, we first describe how the constraint-based STE problem is formulated in [18].

A SAT formula that describes the constraint-based STE problem is built as follows. The variables of the SAT formula include all the symbolic variables in V. In addition, each node (n, t) is represented by two Boolean variables, $v^1_{n,t}$ and $v^0_{n,t}$. Similarly to the dual rail representation in the defining trajectory, the assignments to these two variables represent the four possible values of (n, t) over the quaternary domain $Q \equiv \{1, 0, X, \bot\}$. The constraints of the SAT formula are as follows:

- Each constraint in A on the value of a node (n, t) is translated to a matching constraint in the SAT formula. We denote this set of constraints $cons(A)$.
- An additional constraint is added to the SAT formula to guarantee that at least one of the requirements in C on the value of a node (n, t) does not hold. We denote this constraint $cons(C)$.
- The value of each node (n, t) is constrained to be different from $\bot$ by adding the following constraint: $\neg v^1_{n,t} \vee \neg v^0_{n,t}$. We refer to this set of constraints as consistency constraints.

- Let $val^1_{n,t}, val^0_{n,t}$ denote the dual rail of the value of the node (n, t) as calculated according to the values of its source nodes. The value of each node is constrained to be at least as accurate as $val^1_{n,t}, val^0_{n,t}$, i.e., it is required that $(v^1_{n,t}, v^0_{n,t}) \sqsubseteq (val^1_{n,t}, val^0_{n,t})$. We refer to this set of constraints as $cons(M)$.

  For example, if (n, t) is an AND-node with two source nodes (m_1, t) and (m_2, t), then the following constraints are added:

  1. If (m_1, t) and (m_2, t) equal 1 then (n, t) equals 1. This is guaranteed by adding the constraint $\neg v^1_{m_1,t} \vee \neg v^1_{m_2,t} \vee v^1_{n,t}$.
  2. If (m_1, t) equals 0 then (n, t) equals 0. This is guaranteed by adding the constraint $\neg v^0_{m_1,t} \vee v^0_{n,t}$.
  3. If (m_2, t) equals 0 then (n, t) equals 0. This is guaranteed by adding the constraint $\neg v^0_{m_2,t} \vee v^0_{n,t}$.

  Note that the constraints in $cons(M)$ already take into account the existence of the consistency constraints.

The final SAT formula is combined of the conjunction of the four sets of constraints. A satisfying assignment ϕ to the SAT formula represents a trajectory π that satisfies A but does not satisfy at least one of the requirements in C, and in addition none of the nodes in π equals $\perp$. More formally, let $\phi(V)$ be the assignment of ϕ to the variables in V. Then it holds that $[\phi(V), \pi \models A] = 1$ and $[\phi(V), \pi \models C] \in \{0, X\}$.

The special vacuity problem arises due to the constraints in $cons(M)$. For each internal node (n, t), the SAT formula allows the situation in which its value is more accurate than the value inferred from its source nodes. This causes a special vacuity problem that is demonstrated by Example 3.

Example 3 *We return to Example 1 from Chapter 2 with the following change: we omit the constraint in A on the internal node N3. The resulting STE assertion is as follows: (In1 is 0) $\wedge$ (In3 is v_1) $\Longrightarrow$ $\mathbf{N}$(N6 is 1). A possible satisfying assignment to the SAT formula may contain the assignments $(In1, 0) = 0$, $(In2, 0) = X$, $(In3, 0) = v_1 = 0$ and $(N3, 0) = 1$. However, for all concrete trajectories σ of M so that $\sigma(0)(In1) = 0$ and $\sigma(0)(In3) = 0$ it holds that $\sigma(0)(N3) = 0$. Thus, the counterexample is spurious.*

The similarity between the constraint-based STE vacuity problem and the general STE vacuity problem is that they both stem from a "hidden" assumption that when the value of a node is X in some trajectory π, then it stands both for 0 and for 1, i.e., there exists a concrete trajectory $\pi^1_c \sqsubseteq \pi$ in which the value of this node is 1 and there exists a concrete trajectory $\pi^2_c \sqsubseteq \pi$ in which the value of

this node is 0. This assumption of course is not always true, as demonstrated by Examples 2 and 3. We emphasize that the general vacuity problem also exists in constraint-based STE.

Note that constraint-based STE vacuity may occur even when the user does not impose constraints on internal nodes, as demonstrated by Example 3. In other words, the general STE vacuity problem can be viewed as stemming from an inconsistent STE assertion and the fact that the STE algorithm cannot detect this inconsistency, whereas the constraint-based STE vacuity problem is an inherent problem in the representation of the STE problem. Thus, it may occur even when the STE assertion is consistent. Note also that constraint-based STE vacuity can occur only when a satisfying assignment is found, whereas the general STE vacuity problem may occur both for passed and failed assertions. The reason is that as opposed to constraints in A on internal nodes, a definite value for other internal nodes is not imposed by the SAT formula. If the formula is unsatisfiable, it is in particular unsatisfiable also for all assignments in which the value of an internal node (that is not constrained by A) is exactly as inferred from the values of its source nodes.

A possible solution is to change constraint-based STE by adding constraints to $cons(M)$ forcing the value of each node to be exactly as inferred by the values of its source nodes. In the case of an AND-node (n, t) with two source nodes, this means doubling the number of constraints in $cons(M)$ on the value of (n, t), since we need to add constraints for the cases in which one of the source nodes equals X and the other equals 1, and for the case in which both source nodes equal X. This formulation is referred to in [18] as simulation-based STE. It is used in the SAT-based STE implementation of [25] (though as can be inferred from Section 5.2 there are additional differences in the formulation of the STE problem as a SAT formula between [25] and [18]). The significant reduction in the number of constraints produced in constraint-based STE compared to simulation-based STE was one of the main motivations for its development. It is mentioned in the analysis of the experimental results of [18] that constraint-based STE outperforms simulation-based STE due to the reduction in the number of generated clauses. Thus the proposed solution to the constraint-based STE vacuity problem will transform constraint-based STE into simulation-based STE and cancel the performance enhancement of constraint-based STE.

An alternative solution is to perform vacuity detection for constraint-based STE in a similar way to vacuity detection for STE. As explained before, if the SAT formula is unsatisfiable, then only general STE vacuity detection is required. If the SAT formula is satisfiable, then both general STE vacuity and constraint-based STE vacuity can be detected at the same time as follows.

Given a satisfying assignment sat to the SAT formula, we denote the value assigned to each node (n, t) by $sat(n, t)$. We denote the value of (n, t) that is

inferred by the values of its source nodes in sat by $val(n, t, sat)$. We can build an antecedent A_{new} which reflects all constraints imposed by sat. A_{new} contains the following constraints:

- Each internal node (n, t) for which $sat(n, t) \sqsubseteq val(n, t, sat)$ and $sat(n, t) \neq val(n, t, sat)$ is constrained to $sat(n, t)$.
- Each input (in, t) for which $sat(n, t) \neq X$ is constrained to $sat(n, t)$.

The internal nodes constrained by A_{new} may contain both nodes constrained by the user and nodes constrained by constraint-based STE. Thus, applying either of the general STE vacuity detection algorithms from Chapter 5 for $A_{new} \Longrightarrow C$ will detect both general STE vacuity and constrained based STE vacuity.

Bibliography

[1] 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.

[2] 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.

[3] Derek L. Beatty, Randal E. Bryant, and Carl-Johan H. Seger. Synchronous circuit verification by symbolic simulation: an illustration. In *AUSCRYPT*. MIT Press, 1990.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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 Verification*, 2000.

[10] Edmond M. Clarke, Orna Grumberg, Muralidhar Talupur, and Dong Wang. Making predicate abstraction efficient: How to eliminate redundant predicates. In *CAV'03: Proceedings of Conference on Computer-Aided Verification*, 2003.

[11] 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 Verification*, 2002.

[12] 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.

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

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

[15] Robert P. Kurshan. *Computer-Aided Verification of coordinating processes - the automata theoretic approach*. 1994.

[16] 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.

[17] Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *DAC*, 1997.

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

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

[20] 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.

[21] 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.

[22] 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 Verification*, 2003.

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

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

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

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

[27] 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.

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