Learning Abstractions for Model Checking

Anubhav Gupta

June 2006

CMU-CS-06-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Edmund M. Clarke - chair
Randal E. Bryant
Bruce H. Krogh
Kenneth L. McMillan (Cadence Berkeley Labs)

This research was sponsored by the National Science Foundation under grant no. CCF-0429120, the Semiconductor Research Corporation under agreement nos. 2002-TJ-1027 and 99-TJ-684, and the Defense Advanced Projects Agency under subcontract no. SA2208-23106PG. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: abstraction, boolean satisfiability, bounded model checking, broken trace, decision tree, formal methods, integer linear programming, machine learning, model checking, quantified boolean formula, refinement, unsatisfiable core.

Dedicated to mom, dad and bhai

Abstract

Learning is a process that causes a system to improve its performance through experience. Inductive learning is the process of learning by examples, i.e. the system learns a general rule from a set of sample instances.

Abstraction techniques have been successful in model checking large systems by enabling the model checker to ignore irrelevant details. The aim of abstraction is to identify a small abstract model on which the property holds. Most previous approaches for automatically generating abstract models are based on heuristics combined with the iterative abstraction-refinement loop. These techniques provide no guarantees on the size of the abstract models.

We present an application of machine learning to counterexample-guided abstraction refinement, and to abstraction without refinement. Our work formulates abstraction as an inductive learner that searches through a set of abstract models. The machine learning techniques precisely identify the information in the design that is relevant to the property. Our approach leverages recent advances in boolean satisfiability and integer linear programming techniques. We provide better control on the size of the abstract models, and our approach can generate the smallest abstract model that proves the property.

Most previous work has focused on applying abstraction to model checking, and bounded model checking is used as a subroutine in many of these approaches. We also present an abstraction technique that speeds up bounded model checking.

We have implemented our techniques for the verification of safety properties on hardware circuits using localization abstraction. Our experimental evaluation shows a significant improvement over previous state-of-the-art approaches.

Acknowledgments

I would like to thank my adviser, Edmund Clarke, for his strong guidance and support. His constant encouragement and his faith in me kept me focused and motivated, and carried me through the ups and downs of graduate research. I am very grateful for having had an opportunity to work with him. A special thanks to Martha Clarke for those wonderful group parties, and for taking good care of our research group.

I would also like to thank the other members of my thesis committee - Randy Bryant, Bruce Krogh and Ken McMillan - for their insightful comments and feedback. The summer I spent at Cadence Berkeley Labs with Ken was a great experience. The discussions we had were very helpful towards this thesis.

I want to thank all the members of the model checking group at CMU. In particular, I would like to thank Ofer Strichman. I worked very closely with him for a significant part of my thesis research. I also want to thank all my friends in Pittsburgh for making my stay a memorable experience.

Finally, I would like to thank my mom, my dad, and my brother Saurabh, for their endless love. This thesis is dedicated to them.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The complexity of computer systems is increasing at a tremendous rate, and this growth in complexity is making it harder to ensure that the systems are error-free. Furthermore, computer systems are now used in almost every aspect of our lives, and many of their applications are safety-critical. Thus, it is extremely important that these systems perform as expected. Simulation-based verification techniques cannot achieve the required level of confidence in the performance of these systems, and therefore, application of formal verification techniques is essential.

Model Checking [Clarke and Emerson, 1981; Clarke *et al.*, 1999] is a formal verification technique that automatically decides whether a finite state system satisfies a temporal property by an exhaustive state-space traversal. The temporal logics that model checkers support can specify most of the properties of interest. This thesis focuses on two classes of temporal properties: *safety properties* and *liveness properties*. Intuitively, a safety property states that something bad never happens, while a liveness property states that something good eventually happens.

The major capacity bottleneck for model checking is the state-space size. The use of symbolic Binary Decision Diagram (BDD) [Bryant, 1986] based representations has allowed model checkers to verify systems with several hundred state elements. However, many industrial designs today are at least an order of magnitude larger. Abstraction techniques [Clarke *et al.*, 1994; Cousot and Cousot, 1977] have been very successful in combating the state-explosion problem. The idea behind abstraction is to build a smaller model, called the abstract model, by grouping together multiple states in the concrete (original) model into a single abstract state. The abstract model preserves all the behaviors of the concrete model by allowing a transition from an abstract state $\hat{s}$ to an abstract state $\hat{t}$ if there is some concrete state in $\hat{s}$ that has a concrete transition to some concrete state in $\hat{t}$. This ensures that if the model checker proves a safety or liveness property on the abstract model, the property also holds on the concrete model. However, abstraction can introduce behaviors that violate the property in the abstract model, even when the property holds on the concrete model. Abstraction techniques search over a predefined set of abstract models, looking for one that proves that property and is small in size.

Starting with Kurshan's *localization reduction* [Kurshan, 1995], there has been a lot of work in automatically generating good abstract models [Barner *et al.*, 2002; Chauhan *et al.*, 2002; Clarke *et al.*, 2000, 2003; Das and Dill, 2002; Glusman *et al.*, 2003; Gupta *et al.*, 2003; Jain *et al.*, 2005; Mang and Ho, 2004; Wang *et al.*, 2001, 2003, 2004a]. Many of these techniques follow the *Counterexample-Guided Abstraction-Refinement* (CEGAR) framework [Clarke *et al.*, 2000, 2003]. These techniques start with an initial abstract model and iteratively add more details (*refinement*) to eliminate *spurious* counterexamples, until the property holds on the abstract model or a counterexample is found on the concrete model. The refinement of the abstract model and the search for a concrete counterexample are guided by the abstract counterexamples produced by the model checker.

The field of *machine learning* [Mitchell, 1997] deals with programs that improve their performance through experience. It is an interdisciplinary area that draws on results from statistics, artificial intelligence, probability, information theory, complexity, etc. In recent years, there have been many advances in this area, and machine learning techniques have been used for a wide variety of applications, ranging from data mining of medical records for detecting important correlations to autonomous vehicle navigation systems.

This thesis shows how machine learning techniques can be used in an abstraction-based model checking framework to identify good abstract models. We demonstrate the applicability of these techniques for abstraction both with and without refinement. These techniques increase the capacity of model checking by learning the information in the design that is relevant to the property.

We present an application of *boolean satisfiability* (SAT) and machine learning techniques to the CEGAR framework. In each iteration of the CEGAR loop, we check whether the abstract system satisfies the specification with a standard OBDD-based symbolic model checker. If a counterexample is reported by the model checker, we try to simulate it on the concrete system with a fast SAT-solver. In other words, we generate and solve a SAT instance that is satisfiable if and only if the counterexample is real. If the instance is not satisfiable, we look for the *failure state*, which is the last state in the longest prefix of the counterexample that is still satisfiable. Note that this process cannot be performed with a standard circuit simulator, because the abstract counterexample does not include values for all the inputs in the concrete model.

We use the failure state in order to refine the abstraction. The abstract system has transitions from the failure state that do not exist in the concrete system, and our refinement strategy tries to eliminate these transitions. Our implementation verifies properties of hardware circuits using localization abstraction. Abstraction is performed by selecting a set of state variables and making them *invisible*, i.e., they are treated as inputs. Refinement corresponds to making some variables visible that were previously invisible. It is important to find a small set of variables to make visible in order to keep the size of the abstract state space manageable. This problem can be reduced to a problem of separating two sets of states (abstraction unites concrete states, and therefore refining an abstraction is the opposite operation, i.e., separation of states). For realistic systems, generating these sets is not feasible, both explicitly and symbolically. Moreover, the minimum separation problem is known to be NP-hard [Clarke *et al.*, 2000, 2003]. Instead of enumerating all the states in these sets, we generate samples of these states and learn the separating variables from these samples using Integer Linear Programming and Decision Tree Learning techniques.

Most of the abstraction techniques in literature are based on refinement. The disadvantage of any refinement-based strategy is that once some irrelevant constraint is added to the abstract model, it is not removed in subsequent iterations. As the model checker discovers longer abstract counterexamples, the constraints that were added to eliminate the previous counterexamples might become redundant. Refinement-based techniques do not identify and remove these constraints from the abstract model. This drawback is present in a refinement-based strategy irrespective of the technique that is used to eliminate spurious counterexamples.

We formalize abstraction for model checking as an *inductive learning* [Mitchell, 1997] problem and present an iterative algorithm for abstraction-based model checking that is *not based on refinement*. In order to formulate abstraction as inductive learning, we need some notion of *samples* that constitute the experience available to the learner. We introduce the notion of *broken traces* which capture the *necessary and sufficient* conditions for the existence of an error path in the abstract model. Our abstraction methodology computes the smallest abstract model that *eliminates* all broken traces. This corresponds to the smallest abstract model that can prove the property. The naive method of computing this model by generating and eliminating all broken traces is infeasible because the set of broken traces is too large to enumerate. Instead, we *learn* this model by generating a set of sample broken traces such that the abstract model that eliminates the broken traces in this set also eliminates all other broken traces. Starting with an empty set, we iteratively generate this set of samples. In each iteration of the loop, we compute an abstract model that eliminates all broken traces in the sample set, and then use the counterexample produced by model checking this abstract model to guide the search for new broken traces that are not eliminated by the current abstract model. The loop terminates when no counterexample is present in the abstract model, or a broken trace corresponding to a real bug is generated. We compare this approach with our learning-based CEGAR approach and other state-of-the-art abstraction techniques.

In the last few years, SAT-based Bounded Model Checking (BMC) [Biere *et al.*, 1999] has gained widespread acceptance in industry as a technique for refuting properties with shallow counterexamples. The extreme efficiency of modern SAT-solvers makes it possible to check properties typically up to a depth of a few hundred cycles. There is a very weak correlation between what is hard for standard BDD-based model checking, and what is hard for BMC. It is many times possible to refute properties with the latter that cannot be handled at all by the former.

Most previous work has focused on applying CEGAR to model checking. This thesis presents a CEGAR technique for BMC. Our technique makes BMC faster, as indicated by our experiments. BMC is also used for generating refinements in the Proof-Based Refinement (PBR) framework [Gupta *et al.*, 2003; McMillan and Amla, 2003]. Previous research has shown that CEGAR and PBR are extreme approaches: CEGAR burdens the model checker and PBR burdens the refinement step [Amla and McMillan, 2004] . We show that our technique unifies PBR and CEGAR into an abstraction-refinement framework that can balance the model checking and refinement efforts.

1.1 Thesis Outline

The rest of the thesis is organized as follows:

- *Chapter 2* provides an introduction to the machine learning concepts that are used in the thesis. The contents of this chapter are based on [Mitchell, 1997].
- *Chapter 3* provides a brief introduction to model checking; the use of abstraction in model checking; and SAT-based bounded model checking.
- *Chapter 4* presents an application of inductive learning techniques to the counterexample-guided abstraction-refinement framework. The contents of this chapter are based on [Clarke *et al.*, 2002, 2004].
- *Chapter 5* formalizes abstraction as an inductive learning problem and presents an abstraction-based model checking framework that is not based on refinement. The contents of this chapter are based on [Gupta and Clarke, 2005].
- *Chapter 6* discusses the theoretical complexity of generating the smallest abstract model that can prove the property.
- *Chapter 7* describes a counterexample-guided abstraction-refinement technique for SAT-based bounded model checking. The contents of this chapter are based on [Gupta and Strichman, 2005].
- *Chapter 8* concludes the thesis with a summary of the major contributions and directions for future research.

Chapter 2

Machine Learning

Definition 2.0.1. *Machine learning* is the process that causes a system to improve its performance at a particular task with experience.

2.1 Inductive Learning

Consider a system with the task of classifying objects $x \in X$ into classifications $c \in C$. The performance of this system is characterized by how successful it is in producing the right classification. The experience available to the system is a set S of samples, where each sample is an object with its corresponding classification. The goal of an inductive learner is to infer a classifying function $f : X \to C$ from these samples that can correctly predict the classification for the unobserved objects (Figure 2.1). The learner searches for this function in a set F of candidate functions, which is implicitly defined by the representation being used for these functions.

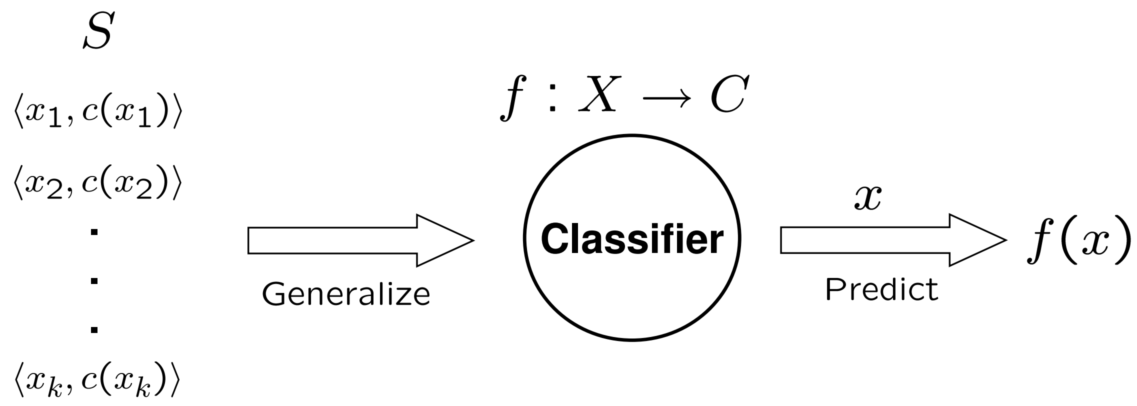

Figure 2.1: Inductive Learning: Generalizing from samples.

Definition 2.1.1. *Inductive learning* is the process of inferring a target concept by generalizing from a set of training samples.

2.2 Generating Samples

There are two frameworks for generating the set of samples:

1. *Random Sampling:* The set of samples is randomly chosen.

2. *Queries:* The learner asks some specific questions about the target function to generate the set of samples.

The types of queries that have been studied in literature [Angluin, 1988] include:

1. *Membership Query:* The input to the query is an object and the output is its classification.

2. *Equivalence Query:* The input to the query is a function g and the output is *yes* if g is the same as the target function f, otherwise the output is an object x and its classification $f(x)$, such that $g(x) \neq f(x)$.

We illustrate these concepts through an example. Consider the problem of learning conjunctions of boolean literals. The set of objects X consists of all assignments to the boolean variables $V = \{x_1, \ldots x_n\}$. The set of classifications C is $\{0, 1\}$. The set of candidate functions F consists of conjunctions of boolean literals over variables in V.

Algorithm 2.1 learns boolean conjunctions with random sampling (LearnConjR). Starting with a boolean conjunction of all the $2n$ literals (line 1), it goes over all samples with classification 1 and removes all literals that are assigned a value 0 in the corresponding object (line 4). If a sample with classification 0 is incorrectly classified by the current function g (line 6), it means that the set of samples S cannot be classified by a conjunction of boolean literals (line 7).

Example 2.2.1. Given $V = \{x_1, x_2, x_3, x_4, x_5\}$, and the sample set

$$S = \{\ \langle (1,0,0,1,1), 1\rangle,\ \langle (1,1,0,1,1), 1\rangle,\ \langle (1,1,0,0,1), 0\rangle,\ \langle (1,1,0,0,0), 1\rangle\ \}$$

We show the working of LearnConjR, as it goes over the samples in S.

1. Initially ($g = x_1 \wedge \neg x_1 \wedge x_2 \wedge \neg x_2 \wedge x_3 \wedge \neg x_3 \wedge x_4 \wedge \neg x_4 \wedge x_5 \wedge \neg x_5$).

2. After $\langle (1,0,0,1,1), 1\rangle$, ($g = x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4 \wedge x_5$).

3. After $\langle (1,1,0,1,1), 1\rangle$, ($g = x_1 \wedge \neg x_3 \wedge x_4 \wedge x_5$).

4. After $\langle(1,1,0,0,1),0\rangle$, $(g = x_1 \wedge \neg x_3 \wedge x_4 \wedge x_5)$.

5. After $\langle(1,1,0,0,0),1\rangle$, $(g = x_1 \wedge \neg x_3)$.

If S also contains the sample $\langle(1,0,0,0,1),0\rangle$, it cannot be classified by a conjunction of boolean literals.

Algorithm 2.1 Learning Boolean Conjunctions with Random Sampling

---

LearnConjR(S)

$g = \bigwedge_{i=1}^{n}(x_i \wedge \neg x_i)$
for $(\langle x, c(x)\rangle \in S$ **do**
    if $(c(x) = 1)$ **then**
        Remove from g all literals that are assigned 0 in s
    else
        if $(g(x) \neq 0)$ **then**
            return *Target function is not a boolean conjunction*
return g

Algorithm 2.2 learns boolean conjunctions with equivalence queries (LearnConjQ). At each iteration, it checks if the current function g matches the target function with an equivalence query (line 3). If g is different from the target function, the sample returned by the query is used to update g (line 8).

Algorithm 2.2 Learning Boolean Conjunctions with Equivalence Queries

LEARNCONJQ

$g = \bigwedge_{i=1}^{n}(x_i \wedge \neg x_i)$
while (1) **do**
  if ($equivalence\_query(g) = yes$) **then**
    return g
  else
    Let $\langle x, c(x) \rangle$ be the result of the query
    if ($c(x) = 1$) **then**
      Remove from g all literals that are assigned 0 in s
    else
      return *Target function is not a boolean conjunction*

2.3 Inductive Bias

In order to be able to meaningfully predict a classification for the unseen objects, an inductive learner must be biased towards certain target functions.

Consider a completely unbiased learner that learns boolean functions by *memorization*. It stores the classification for each sample object in a table. When asked to classify an object, if the object is present in the table, the corresponding classification is returned, otherwise it randomly returns 0 or 1. This example illustrates that an unbiased learner has no rational basis for classifying the unseen objects.

Definition 2.3.1. The *inductive bias* of a learner is the set of assumptions that the learner makes to generalize beyond the samples in the training set.

There are two forms of inductive biases:

1. *Restriction Bias:* This bias is captured by the set F of candidate functions. For an inductive learner with restriction bias, F contains only a subset of the set of all functions from $X \rightarrow C$. The assumption is that the target function is present in F.

2. *Preference Bias:* This bias is captured by the order in which the learner looks at the functions as it searches in F. An inductive learner with a preference bias gives preference to a function f_1 over f_2, even when both f_1 and f_2 classify the training samples. The assumption is that f_1 is a better predictor for the unseen objects.

An inductive learner can also have a combination of the two biases. For example, the LearnConjR and LearnConjQ algorithms have both a restriction bias and a preference bias.

1. Restriction Bias: Instead of all boolean functions of n variables (there are 2^{2^n} such functions), the set F only consists of functions that are conjunctions of boolean literals (there are only 3^n such functions).

2. Preference Bias: These algorithms prefer a function f_1 over f_2 (assuming both f_1 and f_2 classify the training samples) if

$$(f_1(x) = 1) \implies (f_2(x) = 1)$$

Chapter 3

Model Checking and Abstraction

3.1 Model Checking for Safety and Liveness

Model Checking [Clarke and Emerson, 1981; Clarke *et al.*, 1999] is a formal verification technique that automatically decides whether a finite state system satisfies a temporal property by an exhaustive state-space traversal. In this thesis, we focus on two classes of temporal properties: *safety properties* and *liveness properties*. Intuitively, a safety property states that something bad never happens, while a liveness property states that something good eventually happens. For example, consider a system with multiple processes accessing a shared resource. The property that two processes do not access the shared resource at the same time is a safety specification. A desirable liveness property for this system is one that states that if a process requests access to the shared resource, the request is eventually granted.

A system is modeled by a transition system $M = (S, I, R, E, F)$ where:

1. S is the set of states.
2. $I \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is the set of transitions.
4. $E \subseteq S$ is the set of error states.
5. $F \subseteq S$ is the set of good states.

The set E consists of the error states for the safety property, and the set F consists of the good states for the liveness property. If no safety properties are specified, E is the empty set, and if no liveness properties are specified, F is the set of states S.

Given a set W, we use the notation $W(x)$ to denote the fact that $x \in W$. Thus, $I(s)$ denotes that s is an initial state, and $R(s_1, s_2)$ indicates that the transition between the states s_1 and s_2 is in R.

Definition 3.1.1. The *size of a model* $M = (S, I, R, E, F)$ is defined as $|S|$, i.e. the size of its state space.

Definition 3.1.2. A *path* in the transition system $M = (S, I, R, E, F)$ is an infinite sequence of states $\pi = \langle s_0, s_1, \ldots \rangle$, such that $I(s_0)$ and $\forall i \geq 0.\ R(s_i, s_{i+1})$.

A transition system M satisfies the safety property if there is no path in M to an error state. If the safety property is violated, the model checker produces a counterexample

$$C_{\mathcal{S}} = \langle s_0 \ldots s_k \rangle$$

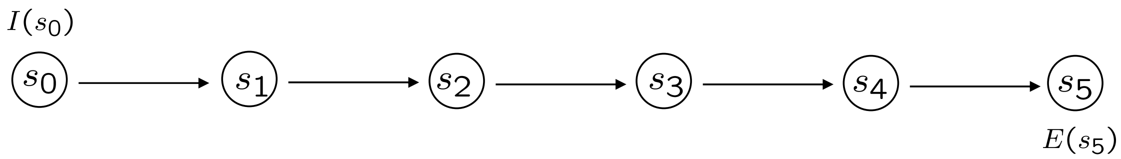

Figure 3.1: A counterexample for a safety property.

which is a prefix of a path in M such that $E(s_k)$ (Figure 3.1).

A transition system M satisfies the liveness property if it can reach a good state on all paths. If the liveness property is violated, there is a path $\pi = \langle s_0, s_1, \ldots \rangle$ in M such that $\forall i \geq 0.\ \neg F(s_i)$. Since the system is finite state, the existence of such a path implies that there exists an infinite repeating path

$$C_{\mathcal{L}} = \langle s_0 \ldots s_{l-1}(s_l \ldots s_k)^{\omega} \rangle$$

which satisfies $\forall 0 \leq i \leq k.\ \neg F(s_i)$. For liveness properties, the counterexample produced by the model checker is such an infinite repeating path (Figure 3.2). In this thesis, we do not handle *fairness* constraints. However, our technique can be easily extended to deal with fairness.

3.2 Abstraction

The major capacity bottleneck for model checking is the size of the model, which can be enormous for real world systems. For example, a simple hardware circuit with 10 32-bit integer registers has a state space size of 2^{320} ($\approx 10^{96}$).

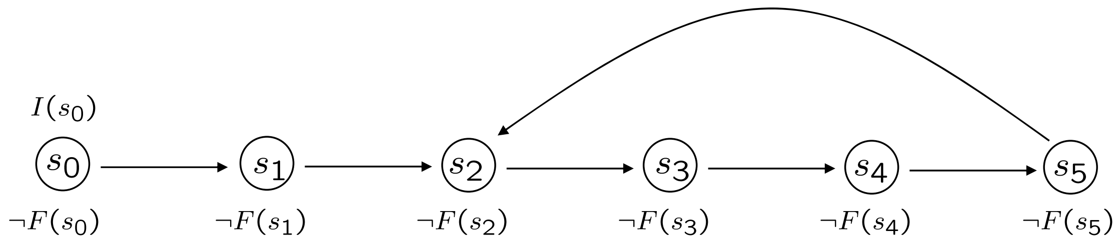

Figure 3.2: A counterexample for a liveness property.

Abstraction [Clarke *et al.*, 1994; Cousot and Cousot, 1977] is one of the most successful techniques to combat this *state-explosion* problem. The idea behind abstraction is to construct a smaller model, called the abstract model, by grouping together multiple states in the concrete model into a single state in the abstract model. The abstract model preserves all the behaviors of the concrete model, which ensures that if the property holds on the abstract model, it also holds on the concrete model. Abstraction works because in many cases, the property at hand does not depend on all the details in the system description. Formally, an abstract model is characterized by an *abstraction function*, which is defined as follows.

Definition 3.2.1. An *abstraction function* $h : S \rightarrow \hat{S}$ for a transition system $M = (S, I, R, E, F)$ maps a concrete state in S to an abstract state in $\hat{S}$.

Given an abstract state $\hat{s}$, $h^{-1}(\hat{s})$ denotes the set of concrete states that are mapped to $\hat{s}$.

Definition 3.2.2. The *minimal abstract model* $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{E}, \hat{F})$ corresponding to a concrete model $M = (S, I, R, E, F)$ and an abstraction function h is defined as follows:

1. $\hat{S} = \{\hat{s} \mid \exists s.\ s \in S \wedge (h(s) = \hat{s})\}$
2. $\hat{I} = \{\hat{s} \mid \exists s.\ I(s) \wedge (h(s) = \hat{s})\}$
3. $\hat{R} = \{(\hat{s}_1, \hat{s}_2) \mid \exists s_1.\ \exists s_2.\ R(s_1, s_2) \wedge (h(s_1) = \hat{s}_1) \wedge (h(s_2) = \hat{s}_2)\}$
4. $\hat{E} = \{\hat{s} \mid \exists s.\ E(s) \wedge (h(s) = \hat{s})\}$
5. $\hat{F} = \{\hat{s} \mid \forall s.\ (h(s) = \hat{s}) \implies F(s)\}$

The above definition corresponds to *existential abstraction.* An abstract state $\hat{s}$ is an error state if there *exists* a concrete error state that is mapped to $\hat{s}$. An abstract state $\hat{s}$ is a good state if *all* concrete states that map to $\hat{s}$ are good states. Minimality implies that $\hat{M}$ has a transition from an abstract state $\hat{s}_1$ to an abstract state $\hat{s}_2$ *only if* there is a state s_1 in $h^{-1}(\hat{s}_1)$ that has a transition in M to a state s_2 in $h^{-1}(\hat{s}_1)$. A non-minimal abstract model, on the other hand, may allow additional transitions. Unless otherwise stated, we will be working with minimal abstract models in this thesis. The essence of abstraction is the following preservation theorem [Clarke *et al.*, 1994].

Theorem 3.2.3. *Let $\hat{M}$ be an abstract model corresponding to M and h. Then if $\hat{M}$ satisfies the safety (liveness) property, then M also satisfies the safety (liveness) property.*

Proof. Let $M = (S, I, R, E, F)$ and $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{E}, \hat{F})$. Assume that M does not satisfy the safety (liveness) property. We show that $\hat{M}$ also violates the safety (liveness) property.

(Safety) Since M does not satisfy the safety property, there is a counterexample $C_{\mathcal{S}} = \langle s_0 \dots s_k \rangle$ on M. Consider the sequence of abstract states $\hat{C}_{\mathcal{S}} = \langle h(s_0) \dots h(s_k) \rangle$ obtained by mapping each state in $C_{\mathcal{S}}$ to its corresponding abstract state. By Definition 3.2.2 we get

1. $I(s_0) \implies \hat{I}(h(s_0))$

2. $\forall 0 \le i < k.\ R(s_i, s_{i+1}) \implies \hat{R}(h(s_i), h(s_{i+1}))$

3. $E(s_k) \implies \hat{E}(h(s_k))$

Thus, $\hat{C}_{\mathcal{S}}$ is a counterexample on $\hat{M}$, which implies that $\hat{M}$ violates the safety property.

(Liveness) Since M does not satisfy the liveness property, there is a counterexample $C_{\mathcal{L}} = \langle s_0 \dots s_{l-1}(s_l \dots s_k)^\omega \rangle$ on M. Consider the sequence of abstract states $\hat{C}_{\mathcal{L}} = \langle h(s_0) \dots h(s_{l-1})(h(s_l) \dots h(s_k))^\omega \rangle$ obtained by mapping each state in $C_{\mathcal{L}}$ to its corresponding abstract state. By Definition 3.2.2 we get

1. $I(s_0) \implies \hat{I}(h(s_0))$

2. $\forall 0 \le i < k.\ R(s_i, s_{i+1}) \implies \hat{R}(h(s_i), h(s_{i+1}))$

3. $R(s_k, s_l) \implies \hat{R}(h(s_k), h(s_l))$

4. $\forall 0 \le i \le k.\ \neg F(s_i) \implies \neg \hat{F}(h(s_i))$

Thus, $\hat{C}_{\mathcal{L}}$ is a counterexample on $\hat{M}$, which implies that $\hat{M}$ violates the liveness property. $\square$

The converse of Theorem 3.2.3 is not true. The property might not hold on the abstract model, even though the property holds on the concrete model. In this case, the abstract counterexample generated by the model checker is *spurious*, i.e. it does not correspond to a concrete path. The current abstract model is too coarse to validate the specification. The aim of abstraction is to identify an abstract model that on the one hand is small enough to be handled by the model checker, and on the other hand avoids grouping of concrete states that introduces spurious counterexamples. Abstraction techniques search for this abstract model over a set of candidate abstract models, which is implicitly defined by the techniques being used to generate these models.

3.3 Abstraction Functions

The number of ways to partition a set with m elements into disjoint non-empty subsets is given by the *Bell number* B_m.

$$B_m = \frac{1}{e}\sum_{k=0}^{\infty}\frac{k^m}{k!}$$

An abstract model essentially corresponds to a partition of the set of concrete states into disjoint non-empty sets. Thus, the number of possible abstract models for $M = (S, I, R, E, F)$ is given by $B_{|S|} \gg 2^{|S|}$. Most abstraction techniques work with only a small part of this large space of all possible abstract models.

3.3.1 Predicate Abstraction

Predicate Abstraction [Graf and Saidi, 1997] has emerged as a popular technique for creating abstract models. Consider a system with a set of state variables $V = \{x_1, \ldots x_n\}$, where each variable x_i ranges over a non-empty domain D_{x_i}. Each state s of the system assigns values to the variables in V. The set of states for the system is $S = D_{x_1} \times \cdots \times D_{x_n}$.

In predicate abstraction, the abstraction functions are characterized by a set of predicates $\mathcal{P} = \{p_1, \ldots p_k\}$ over the variables V. The predicates are formulas from some predefined theory (for example: separation logic, linear arithmetic, etc). The abstract model has a boolean state variable b_i corresponding to each predicate p_i. Each abstract state $\hat{s}$ assigns values to the variables $\hat{V} = \{b_1, \ldots b_k\}$. The value of the variable b_i in state $\hat{s}$ indicates the truth value of the corresponding predicate in that state. The set of abstract states is $\hat{S} = \{0, 1\}^k$. The abstraction function $h^{\mathcal{P}}$ corresponding to the set of predicates $\mathcal{P}$ is given by

$$h^{\mathcal{P}}(s) = \bigwedge_{i=1}^{k} (p_i(s) \iff b_i) \tag{3.1}$$

This abstraction functions maps a concrete state s on to the abstract state corresponding to the valuation of the predicates at s. An abstraction technique based on predicate abstraction only considers abstraction functions of the form $h^{\mathcal{P}}$, where $\mathcal{P}$ is a finite set of predicates from the theory of interest.

3.3.2 Localization Abstraction

Localization abstraction [Kurshan, 1995] is another widely used technique for creating abstract models, especially for the verification of hardware circuits. Localization abstraction partitions the set of state variables V into two sets: the set of *visible* variables which we denote by $\mathcal{V}$ and the set of *invisible* variables which we denote by $\mathcal{I}$. Let $s(x)$, $x \in V$ denote the value of variable x in a state s. Given a set of variables $U = \{u_1, \ldots u_p\}$, $U \subseteq V$, s^U denotes the portion of s that corresponds to the variables in U, i.e. $s^U = (s(u_1) \ldots s(u_p))$. Let $\mathcal{V} = \{v_1, \ldots v_k\}$. The abstract model consists of only the visible variables, i.e. the set of abstract states is $\hat{S} = D_{v_1} \times \cdots \times D_{v_k}$. The abstraction function $h^{\mathcal{V}}$ corresponding to the set of visible variables $\mathcal{V}$ is given by

$$h^{\mathcal{V}}(s) = s^{\mathcal{V}} \tag{3.2}$$

This abstraction functions maps a concrete state s on to the abstract state corresponding to the assignment to the visible variables at s. An abstraction technique based on localization abstraction only considers abstraction functions of the form h^U, where $U \subseteq V$.

3.3.3 Abstract Models for Localization Abstraction

Definition 3.3.1. Given a model $M = (S, I, R, E, F)$, the transition relation $R(s_1, s_2)$ is said to be in *functional form* if the next state s_2 is specified as a function of the current state s_1. Formally, $R(s_1, s_2)$ is of the form

$$R(s_1, s_2) \;=\; (s_2 = G(s_1))$$

where G is some function of s_1. The function G does not depend on s_2.

For an arbitrary system M and abstraction function h, it is often too expensive to construct the minimal abstract model $\hat{M}$ [Clarke *et al.*, 1994], because computation of this model requires exact quantification over the state variables in M (Definition 3.2.2). For localization abstraction, on the other hand, we can compute $\hat{M}$ efficiently for systems where the transition relation R is in functional form. For these systems, $\hat{M}$ can be computed syntactically from the system description, by removing the logic that defines the invisible variables and treating them as inputs. This construction also explains why this technique is called localization: because it *localizes* the part of the system description which is responsible for the ensuring that the property holds.

Theorem 3.3.2. *If the transition relation is in functional form, the model obtained by removing from R the logic that defines the invisible variables $\mathcal{V}$ and replacing them with non-deterministic inputs, represents the minimal abstract model corresponding to the abstraction function $h(s) = s^{\mathcal{V}}$.*

Proof. The proof is based on the observation that we can quantify out the next-state copy of the invisible variables if the transition relation is in functional form, because in that case, the value of each next-state variable does not depend on other next-state variables.

Consider a model with state variables $\{x_1, \ldots x_n\}$ and inputs $\{i_1, \ldots i_q\}$. We use the standard notation x' to denote the next-state version of x. Let $s = (x_1, \ldots x_n)$, $s' = (x'_1, \ldots x'_n)$ and $i = (i_1, \ldots i_q)$. Since the transition relation R is in functional form, it can be expressed as:

$$R(s, s') = \exists i.\ (\ \bigwedge_{j=1}^{n} x'_j = f_{x_j}(s, i)\ )$$

where f_{x_j} is the functional definition of x'_j. By Definition 3.2.2, the minimal abstract transition relation $\hat{R}$ is given by:

$$\hat{R}(\hat{s}, \hat{s}') = \exists s.\ \exists s'.\ (\ R(s, s')\ \wedge\ h(s) = \hat{s}\ \wedge\ h(s') = \hat{s}'\ )$$

Substituting expressions for R and the abstraction function h, and splitting s and s' into visible and invisible parts yields

$$\begin{aligned}\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{V}}.\ \exists s^{\mathcal{I}}.\ \exists s'^{\mathcal{V}}.\ \exists s'^{\mathcal{I}}.\ \exists i.\ (\ &\bigwedge_{x_j \in \mathcal{V}} x'_j = f_{x_j}(s^{\mathcal{V}}, s^{\mathcal{I}}, i)\ \wedge\ \bigwedge_{x_j \in \mathcal{I}} x'_j = f_{x_j}(s^{\mathcal{V}}, s^{\mathcal{I}}, i) \\ &\wedge\ s^{\mathcal{V}} = \hat{s}\ \wedge\ s'^{\mathcal{V}} = \hat{s}'\ )\end{aligned}$$

We can eliminate the quantification over the visible variables using the rule:

$$\exists a.\ (\ f(a)\ \wedge\ a = b\ )\ \equiv\ f(b)$$

This gives us

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}}.\ \exists s'^{\mathcal{I}}.\ \exists i.\ (\ \bigwedge_{x_j \in \mathcal{V}} \hat{x}'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i)\ \wedge\ \bigwedge_{x_j \in \mathcal{I}} x'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i)\ )$$

Since the left conjunct does not depend on $s'^{\mathcal{I}}$, we can push the quantification over $s'^{\mathcal{I}}$ to the right conjunct, which gives us

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}}.\ \exists i.\ (\ \bigwedge_{x_j \in \mathcal{V}} \hat{x}'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i)\ \wedge\ \exists s'^{\mathcal{I}}.\ (\ \bigwedge_{x_j \in \mathcal{I}} x'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i)\ )\ )$$

The quantification over $s'^{\mathcal{I}}$ evaluates to TRUE because f_{x_j} does not depend on x'_j. Thus, the expression simplifies to

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}}.\ \exists i.\ (\ \bigwedge_{x_j \in \mathcal{V}} \hat{x}'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i)\ )$$

which corresponds to the transition relation derived syntactically from the original R by replacing the invisible variables with inputs. $\square$

Sometimes, constraints are imposed on the system during verification. These constraints may arise, for example, from the restrictions on the environment or from the don't care space. In these cases, the transition relation has a non-functional component g. If g is a function of only the present state variables and the inputs, the minimal abstract transition relation is given by

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}}.\ \exists i.\ (\ \bigwedge_{x_j \in \mathcal{V}} \hat{x}'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i)\ \wedge\ g(\hat{s}, s^{\mathcal{I}}, i)\ )$$

Intuitively, in addition to the logic for the visible variables, the constraints are also included in the minimal abstract model.

In some scenarios, the non-functional component g might also depend on some next-state variables. Let $\mathcal{T} \subseteq \mathcal{I}$ denote the set of next-state invisible variables that are present in g. For these models, the minimal abstract transition relation is given by

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}}.\ \exists i.\ \exists s'^{\mathcal{T}}.\ (\ \bigwedge_{x_j \in \mathcal{V}} \hat{x}'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i)\ \wedge\ \bigwedge_{x_j \in \mathcal{T}} \hat{x}'_j = f_{x_j}(\hat{s}, s^{\mathcal{I}}, i)$$

$$\wedge\ g(\hat{s}, s^{\mathcal{I}}, s'^{\mathcal{I}}, i)\ )$$

In addition to the logic for the visible variables and the constraints, the logic defining the next-state invisible variables present in these constraints is also included in the minimal abstract model. The proof for this construction is similar to the proof for Theorem 3.3.2.

3.3.4 Pros and Cons of Localization Abstraction

In this thesis, we will use localization abstraction to construct the abstract models. Localization abstraction has the following advantages:

1. The minimal abstract models can be efficiently constructed (Theorem 3.3.2).

2. Compared to approaches like predicate abstraction, the space of possible abstract models is smaller. This makes it easier to identify an abstract model that proves the property.

However, this restriction on the space of candidate abstract models can also be viewed as a disadvantage. Localization abstraction will work only if the property at hand is *localizable,* i.e. its validity depends on a small part of the system description. For example, consider a property that depends on the fact that the two state variables x_1 and x_2 are always equal. To be able to prove this property, a framework based on localization abstraction would require that both x_1 and x_2 are included in the abstract model. This might make the abstract model intractable for the model checker. Using predicate abstraction, this property can be proved by a smaller abstract model that only contains the boolean variable b corresponding to the predicate $(x = y)$. As discussed in Chapter 8, the techniques presented in this thesis can also be used with predicate abstraction.

3.4 An Example

Consider the model $\mathcal{M}$ in Figure 3.3, which is shown in NuSMV [Cimatti *et al.*, 2002] input format. It has 7 boolean state variables, namely x, y, z, u, $c_0.v$, $c_1.v$ and $c_2.v$; and i,j are boolean inputs. The state space of $\mathcal{M}$ is $S = \{0,1\}^7$. The variables $c_0.v$, $c_1.v$ and $c_2.v$ implement a 3-bit counter. The variable u is updated every time the counter hits the value 7. The model has a safety specification which states that u is always 1. Note that this property holds on the model. All the logic is in the cone of influence of the property, so the cone of influence reduction will not help in reducing the size of the model for model checking. We will be using this example later in this thesis as well.

Consider the localization abstraction function $h_1 : \{0,1\}^7 \rightarrow \{0,1\}^3$, for the set of visible variables $\mathcal{V} = \{x, y, u\}$.

$$h_1(\ (x, y, z, u, c_0.v, c_1.v, c_2.v)\ ) = (x, y, u) \tag{3.3}$$

The abstract model corresponding to $\mathcal{M}$ and h_1 is shown in Figure 3.4. This model has been derived using the construction described in Section 3.3.3. The property holds on this model, and therefore, by Theorem 3.2.3, the property also holds on $\mathcal{M}$.

Another localization of $\mathcal{M}$ that proves this property corresponds to $\mathcal{V} = \{z, c_0.v, u\}$.

```
MODULE main
VAR
    x, y, z, u : boolean;
    c0 : counter_cell(1);
    c1 : counter_cell(c0.cout);
    c2 : counter_cell(c1.cout);
IVAR
    i, j : boolean;
ASSIGN
    init(x) := j;
    next(x) := i;

    init(y) := !j;
    next(y) := !i;

    init(z) := 0;
    next(z) := !z;

    init(u) := 1;
    next(u) := case
                 c0.v & c1.v & c2.v : x | y | z;
                 1 : u;
               esac;
SPEC AGu

MODULE counter_cell(cin)
VAR
    v : boolean;
ASSIGN
   init(v) := 0;
   next(v) := v + cin mod 2;
DEFINE
   cout := v & cin;
```

Figure 3.3: Description of model $\mathcal{M}$ in NuSMV input format.

```
MODULE main
VAR
        x, y, u : boolean;
IVAR
        i, j, z, c0, c1, c2 : boolean;
ASSIGN
        init(x) := j;
        next(x) := i;

        init(y) := !j;
        next(y) := !i;

        init(u) := 1;
        next(u) := case
                      c0 & c1 & c2 : x | y | z;
                      1 : u;
                     esac;
SPEC AG u
```

Figure 3.4: Abstract model corresponding to $\mathcal{M}$ (Figure 3.3) and h_1 (Equation 3.3).

The abstract model is shown in Figure 3.5.

On the other hand, the abstract model for $\mathcal{V} = \{y, z, u\}$ (Figure 3.6) does not prove the property. Model checking of this model produces a spurious counterexample which is shown in Figure 3.7.

3.5 Bounded Model Checking

Bounded Model Checking (BMC) [Biere *et al.*, 1999] is a powerful technique for refuting properties. The basic idea in BMC is to search for a counterexample in executions of some bounded depth. The depth is increased until either a counterexample is found or the problem becomes too hard to solve. Given a model $M = (S, I, R, E, F)$, and a positive integer k representing the depth of the search, BMC generates the formulas $M^k_{\mathcal{S}}$ (Equation 3.4) and $M^k_{\mathcal{L}}$ (Equation 3.5), for checking the safety and liveness property, respectively. These formulas are satisfiable if and only if there is a counterexample of length k for the respective property.

$$M^k_{\mathcal{S}} = I(s_0) \ \wedge \ \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \ \wedge \ E(s_k) \tag{3.4}$$

$$M^k_{\mathcal{L}} = I(s_0) \ \wedge \ \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \ \wedge \ \bigwedge_{i=0}^{k} \neg F(s_i) \ \wedge \ \bigvee_{i=0}^{k} R(s_k, s_i) \tag{3.5}$$

The satisfiability of these formulas can be efficiently reduced to *Boolean Propositional Satisfiability* (SAT). The extreme efficiency of modern SAT-solvers makes it possible to check properties typically up to a depth of a few hundred cycles. There is a very

```
MODULE main
VAR
      z, u : boolean;
      c0 : counter_cell(1);
IVAR
      i, j, x, y, c1, c2 : boolean;
ASSIGN
      init(z) := 0;
      next(z) := !z;
      init(u) := 1;
      next(u) := case
                   c0.v & c1 & c2 : x | y | z;
                   1 : u;
                 esac;
SPEC AGu
MODULE counter_cell(cin)
VAR
      v : boolean;
ASSIGN
     init(v) := 0;
     next(v) := v + cin mod 2;
DEFINE
     cout := v & cin;
```

Figure 3.5: Abstract model for $\mathcal{M}$ (Figure 3.3) with $\mathcal{V} = \{z, c_0.v, u\}$.

```
MODULE main
VAR
        y, z, u : boolean;
IVAR
        i, j, x, c0, c1, c2 : boolean;
ASSIGN
        init(y) := !j;
        next(y) := !i;

        init(z) := 0;
        next(z) := !z;

        init(u) := 1;
        next(u) := case
                      c0 & c1 & c2 : x | y | z;
                      1 : u;
                    esac;
SPEC AGu
```

Figure 3.6: Abstract model for $\mathcal{M}$ (Figure 3.3) with $\mathcal{V} = \{y, z, u\}$.

State: (y,z,u)

$$(0,0,1) \longrightarrow \underset{\neg u}{(0,1,0)}$$

Figure 3.7: A counterexample on the model in Figure 3.6.

weak correlation between what is hard for standard BDD-based model checking, and what is hard for BMC. It is many times possible to refute properties with the latter that cannot be handled at all by the former.

Chapter 4

Abstraction-Refinement and Learning

4.1 Abstraction-Refinement

Definition 4.1.1. Given a transition system $M = (S, I, R, E, F)$ and an abstraction function h, h' is a *refinement* of h (denoted by $h \prec h'$) if

1. For all $s_1, s_2 \in S$, $h'(s_1) = h'(s_2)$ implies $h(s_1) = h(s_2)$.

2. There exists $s_1, s_2 \in S$ such that $h(s_1) = h(s_2)$ and $h'(s_1) \neq h'(s_2)$.

If $h \prec h'$, then the minimal abstract model corresponding to M and h' is also called a *refinement* of the minimal abstract model corresponding to M and h.

For example, for localization abstraction, given two sets of variables $\mathcal{V}_1$ and $\mathcal{V}_2$,

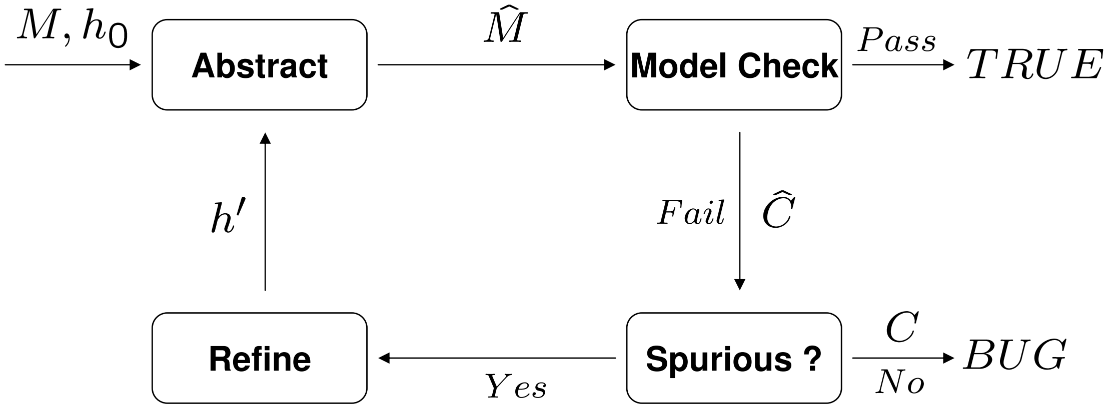

Figure 4.1: Counterexample-Guided Abstraction-Refinement (CEGAR).

if $\mathcal{V}_1 \subset \mathcal{V}_2$ then $h^{\mathcal{V}_1} \prec h^{\mathcal{V}_2}$ (Equation 3.2).

There has been a lot of work on automatically generating abstract models that prove the property. Many of these techniques follow the *Counterexample-Guided Abstraction-Refinement* (CEGAR) framework [Clarke *et al.*, 2000, 2003]. These techniques start with an initial abstraction function and iteratively refine it to eliminate spurious counterexamples, until the property holds on the abstract model or a counterexample is found on the concrete model (Figure 4.1). The refinement of the abstract model and the search for a concrete counterexample are guided by the abstract counterexamples produced by the model checker (hence the name Counterexample-Guided).

In the rest of the chapter, we present an implementation of the CEGAR loop that uses SAT-solvers and machine learning techniques for the verification of hardware circuits.

4.2 Step: Abstract

We use localization abstraction based on visible/invisible variables (Section 3.3.2). Thus, we use the syntactic construction described in Section 3.3.3 to build the abstract models. Our initial abstract model corresponds to $\mathcal{V} = \{\}$, i.e. all state variables are invisible.

4.3 Step: Model Check

We can use any off-the-shelf checker for this step. Our implementation interfaces with two state-of-the-art model checkers, Cadence SMV [McMillan] and NuSMV [Cimatti *et al.*, 2002].

4.4 Step: Spurious?

For both safety and liveness properties, the counterexample is a finite sequence of abstract states $\hat{s}_0, \ldots \hat{s}_k$ (Section 3.1). Given a counterexample C on the abstract model corresponding to an abstraction function h, and $0 \leq i \leq k$, and $0 \leq j \leq k$, let $C^i(s_j)$ denote the fact that the concrete state s_j is mapped to the counterexample state $\hat{s}_i$ by the abstraction function h:

$$C^i(s_j) \;=\; (h(s_j) = \hat{s}_i) \tag{4.1}$$

As explained in Section 3.3.2, $h(s_j)$ is a projection of s_j on to the visible variables.

Therefore, $C^i(s_j)$ is simply a restriction of the visible variables in state s_j to their values in the counterexample state $\hat{s}_i$.

For safety properties, the counterexample generated by the model checker is a prefix to a path such that the last state in the prefix is an error state (Section 3.1):

$$C_{\mathcal{S}} = \langle \hat{s}_0, \ldots \hat{s}_k \rangle$$

The set of concrete error traces that agree with $C_{\mathcal{S}}$ is given by:

$$\psi_{C_{\mathcal{S}}} = \{ \ \langle s_0, \ldots s_k \rangle \ | \ \ I(s_0) \ \wedge \ \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \ \wedge \ E(s_k) \ \wedge \ \bigwedge_{i=0}^{k} C^i_{\mathcal{S}}(s_i) \ \} \tag{4.2}$$

For liveness properties, the counterexample produced by the model checker is an infinite repeating path (Section 3.1):

$$C_{\mathcal{L}} = \langle \hat{s}_0 \ldots \hat{s}_{l-1} (\hat{s}_l \ldots \hat{s}_k)^{\omega} \rangle$$

The set of concrete paths that agree with $C_{\mathcal{L}}$ is given by:

$$\begin{aligned} \psi_{C_{\mathcal{L}}} = \{ \ \langle s_0 \ldots s_{l-1} (s_l \ldots s_k)^{\omega} \rangle \ | \ I(s_0) \ \wedge \ \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \ \wedge \ R(s_k, s_l) \ \wedge \\ \bigwedge_{i=0}^{k} \neg F(s_i) \ \wedge \ \bigwedge_{i=0}^{k} C^i_{\mathcal{L}}(s_i) \ \} \end{aligned} \tag{4.3}$$

The counterexample is spurious if and only if the corresponding set ψ is empty. We check for that by solving ψ with a SAT-solver. This formula is very similar in structure to the formulas that arise in Bounded Model Checking (BMC) (Section 3.5). However, ψ is easier to solve because the traces are restricted to agree with the counterexample. Most model checkers treat inputs as state variables, and therefore the counterexample includes assignments to inputs. While simulating the counterexample, we also restrict the values of the (original) inputs that are part of the definition (lie on the RHS) of the visible variables to the value assigned to them by the counterexample, which further simplifies the formula. If a satisfying assignment is found, it corresponds to a real bug and the property fails on the model.

4.5 Step: Refine

If the counterexample is spurious, we refine the abstract model to eliminate this spurious behavior. Our approach is based on the work by Yuan Lu et al. [Clarke *et al.*, 2000, 2003].

4.5.1 Safety Properties

Given a spurious counterexample $C_{\mathcal{S}} = \langle \hat{s}_0, \ldots \hat{s}_k \rangle$ for a safety property, consider the following two sequences of formulas:

$$\hat{\mathcal{D}}^i_{\mathcal{S}} = \begin{cases} I(s_0) \ \wedge \ C^0_{\mathcal{S}}(s_0) & i = 0 \\ \hat{\mathcal{D}}^{i-1}_{\mathcal{S}} \ \wedge \ R(s_{i-1}, s_i) \ \wedge \ C^i_{\mathcal{S}}(s_i) & 0 < i \leq k \end{cases} \tag{4.4}$$

$$\hat{\mathcal{B}}^i_{\mathcal{S}} = \begin{cases} C^i_{\mathcal{S}}(s_i) \ \wedge \ R(s_i, s_{i+1}) \ \wedge \ C^{i+1}_{\mathcal{S}}(s_{i+1}) & 0 \leq i < k \\ C^k_{\mathcal{S}}(s_k) \ \wedge \ E(s_k) & i = k \end{cases} \tag{4.5}$$

These definitions satisfy:

$$\hat{\mathcal{D}}_{\mathcal{S}}^{i+1} \;=\; \hat{\mathcal{D}}_{\mathcal{S}}^{i} \;\wedge\; \hat{\mathcal{B}}_{\mathcal{S}}^{i} \quad (0 \leq i < k) \tag{4.6}$$

$$\hat{\mathcal{D}}_{\mathcal{S}}^{k} \;\wedge\; \hat{\mathcal{B}}_{\mathcal{S}}^{k} \;=\; \psi_{C_{\mathcal{S}}} \tag{4.7}$$

Let $\mathcal{D}_{\mathcal{S}}^{i}(s_i)$ denote the restriction of $\hat{\mathcal{D}}_{\mathcal{S}}^{i}$ to the s_i variables.

$$\mathcal{D}_{\mathcal{S}}^{i}(s_i) \;=\; \hat{\mathcal{D}}_{\mathcal{S}}^{i} \downarrow s_i \;\equiv\; \exists s_0 \ldots s_{i-1}.\ \hat{\mathcal{D}}_{\mathcal{S}}^{i} \tag{4.8}$$

Similarly, let $\mathcal{B}_{\mathcal{S}}^{i}(s_i)$ denote the restriction of $\hat{\mathcal{B}}_{\mathcal{S}}^{i}$ to the s_i variables.

$$\mathcal{B}_{\mathcal{S}}^{i}(s_i) \;=\; \hat{\mathcal{B}}_{\mathcal{S}}^{i} \downarrow s_i \;\equiv\; \exists s_{i+1}.\ \hat{\mathcal{B}}_{\mathcal{S}}^{i} \tag{4.9}$$

We find the largest index f such that $\mathcal{D}_{\mathcal{S}}^{f}$ is satisfiable. The index f is called the *failure index*, and the abstract state $\hat{s}_f$ is called the *failure state*. The set of concrete states s_f that is characterized by the formula $\mathcal{D}_{\mathcal{S}}^{f}(s_f)$ is called the set of *deadend states*. We denote this set by D. The set D consists of all states s_f such that there exists some concrete trace $\langle s_1, \ldots s_f \rangle$ that agrees with the counterexample $C_{\mathcal{S}}$, i.e.

$$\forall 0 \leq i \leq f.\ C_{\mathcal{S}}^{i}(s_i)$$

The set of concrete states s_f that is characterized by the formula $\mathcal{B}_{\mathcal{S}}^{f}(s_f)$ is called the set of *bad states*. We denote this set by B. Both D and B are mapped to the failure state in the abstract model. Since f is the largest index for which $\mathcal{D}_{\mathcal{S}}^{f}$ is satisfiable, Equation 4.6 implies that

$$D \cap B = \{\} \tag{4.10}$$

The counterexample is spurious for one of two different reasons, which depends on the value of f. We illustrate the two scenarios, corresponding to the two cases in Equation 4.5:

1. $0 \leq f < k$: (Figure 4.2) The counterexample is spurious because there is no concrete transition from a state in D, to a concrete state in $h^{-1}(\hat{s}_{f+1})$. Since there is an abstract transition from $\hat{s}_f$ to $\hat{s}_{f+1}$, there is a non-empty set of concrete transitions from $h^{-1}(\hat{s}_f)$ to $h^{-1}(\hat{s}_{f+1})$ (Definition 3.2.2). The set B consists of the concrete states in $\hat{s}_f$ that have a transition to a concrete state in $\hat{s}_{f+1}$.

2. $f = k$: (Figure 4.3) The counterexample is spurious because there is no error state in D. Since s_k is an error state in the abstract model, there is a non-empty set of concrete error states in $h^{-1}(\hat{s}_k)$ (Definition 3.2.2). The set B refers to this set of error states.

4.5.2 Liveness Properties

In order to simplify our discussion on liveness properties, we assume that our initial abstract model *respects* the non-good states, i.e. an abstract state is labeled as a non-good state only if all the concrete states that map to that abstract state are non-good states. Formally:

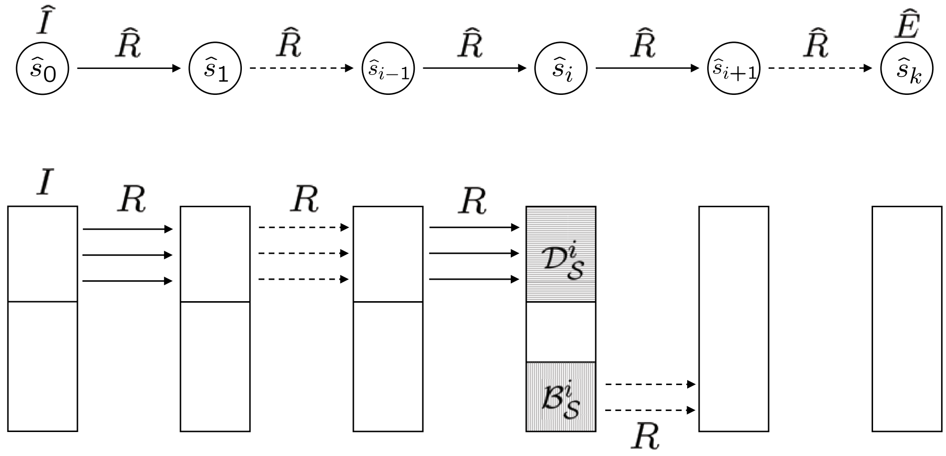

Figure 4.2: (Safety property) The counterexample is spurious because there is no concrete transition from a deadend state to the next abstract state.

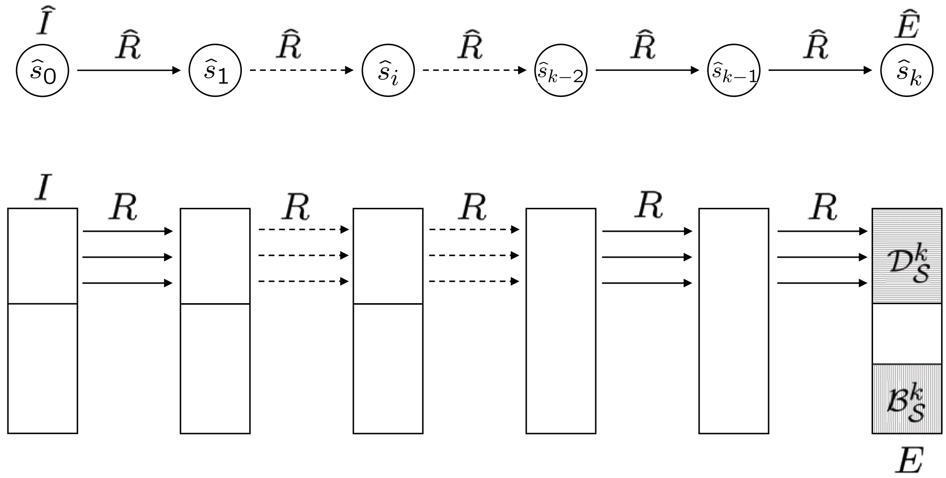

Figure 4.3: (Safety property) The counterexample is spurious because there are no error states in the set of deadend states.

$$\neg F(\hat{s}) \;\wedge\; (h(s) = (\hat{s})) \implies \neg F(s) \tag{4.11}$$

Since the initial abstract model respects the non-good states, the abstract models generated in the subsequent iterations of the CEGAR loop, which are refinements of the initial abstract model, also respect the non-good states.

Given a spurious counterexample $C_{\mathcal{L}} = \langle \hat{s}_0 \ldots \hat{s}_{l-1}(\hat{s}_l \ldots \hat{s}_k)^{\omega} \rangle$ for a liveness property, consider the following two sequences of formulas:

$$\hat{\mathcal{D}}^i_{\mathcal{L}} \;=\; \begin{cases} I(s_0) \;\wedge\; C^0_{\mathcal{L}}(s_0) & i = 0 \\ \hat{\mathcal{D}}^{i-1}_{\mathcal{L}} \;\wedge\; R(s_{i-1}, s_i) \;\wedge\; C^i_{\mathcal{L}}(s_i) & 0 < i \leq k \\ \hat{\mathcal{D}}^k_{\mathcal{L}} \;\wedge\; R(s_k, s_{k+1}) \;\wedge\; C^l_{\mathcal{L}}(s_{k+1}) & i = k+1 \end{cases} \tag{4.12}$$

$$\hat{\mathcal{B}}^i_{\mathcal{L}} \;=\; \begin{cases} C^i_{\mathcal{L}}(s_i) \;\wedge\; R(s_i, s_{i+1}) \;\wedge\; C^{i+1}_{\mathcal{L}}(s_{i+1}) & 0 \leq i < k \\ C^k_{\mathcal{L}}(s_k) \;\wedge\; R(s_k, s_{k+1}) \;\wedge\; C^l_{\mathcal{L}}(s_{k+1}) & i = k \\ \hat{\mathcal{D}}^{k+1}_{\mathcal{L}}[s_w/s_{k+1}][s_{k+1}/s_l] & i = k+1 \end{cases} \tag{4.13}$$

where $g[a/b]$ denotes the formula obtained by substituting a for b in the formula g. These definitions satisfy:

$$\hat{\mathcal{D}}^{i+1}_{\mathcal{L}} \;=\; \hat{\mathcal{D}}^i_{\mathcal{L}} \;\wedge\; \hat{\mathcal{B}}^i_{\mathcal{L}} \quad (0 \leq i < k+1) \tag{4.14}$$

Moreover, since $C_{\mathcal{L}}$ is a counterexample for the liveness property, it satisfies $\forall 0 \leq i \leq k.\ \neg F(\hat{s}_i)$. Therefore, using Equation 4.11, we get

$$\hat{\mathcal{D}}_{\mathcal{L}}^{k+1} \ \wedge \ \hat{\mathcal{B}}_{\mathcal{L}}^{k+1} \ = \ \hat{\mathcal{D}}_{\mathcal{L}}^{k+1} \ \wedge \ \hat{\mathcal{B}}_{\mathcal{L}}^{k+1} \ \wedge \ \bigwedge_{i=0}^{k} \neg F(s_k) \ = \ \psi_{C_{\mathcal{L}}} \tag{4.15}$$

Let $\mathcal{D}_{\mathcal{L}}^{i}(s_i)$ denote the restriction of $\hat{\mathcal{D}}_{\mathcal{L}}^{i}$ to the s_i variables.

$$\mathcal{D}_{\mathcal{L}}^{i}(s_i) \ = \ \hat{\mathcal{D}}_{\mathcal{L}}^{i} \downarrow s_i \ \equiv \ \exists s_0 \ldots s_{i-1}. \ \hat{\mathcal{D}}_{\mathcal{L}}^{i} \tag{4.16}$$

Similarly, let $\mathcal{B}_{\mathcal{L}}^{i}(s_i)$ denote the restriction of $\hat{\mathcal{B}}_{\mathcal{L}}^{i}$ to the s_i variables.

$$\mathcal{B}_{\mathcal{L}}^{i}(s_i) \ = \ \hat{\mathcal{B}}_{\mathcal{L}}^{i} \downarrow s_i \ \equiv \ \exists s_0 \ldots s_{i-1} s_{i+1} s_w. \ \hat{\mathcal{B}}_{\mathcal{L}}^{i} \tag{4.17}$$

We find the largest index f such that $\mathcal{D}_{\mathcal{L}}^{f}$ is satisfiable. The index f is called the *failure index*. The set of concrete states s_f that is characterized by the formula $\mathcal{D}_{\mathcal{L}}^{f}(s_f)$ is called the set of *deadend states*. We denote this set by D. The set D consists of all states s_f such that there exists some concrete trace $\langle s_1, \ldots s_f \rangle$ that agrees with the counterexample $C_{\mathcal{L}}$, i.e.

$$\forall 0 \leq i \leq f. \ C_{\mathcal{L}}^{i}(s_i)$$

The set of concrete states s_f that is characterized by the formula $\mathcal{B}_{\mathcal{L}}^{f}(s_f)$ is called the set of *bad states*. We denote this set by B. Both D and B are mapped to the failure state in the abstract model. Since f is the largest index for which $\mathcal{D}_{\mathcal{L}}^{f}$ is satisfiable, Equation 4.14 implies that

$$D \cap B = \{\} \tag{4.18}$$

The counterexample is spurious for one of three different reasons, which depends on the value of f. We illustrate the three scenarios, corresponding to the three cases in Equation 4.13:

1. $0 \leq f < k$: (Figure 4.4) The counterexample is spurious because there is no concrete transition from a state in D to a concrete state in $h^{-1}(\hat{s}_{f+1})$. Since there is an abstract transition from $\hat{s}_f$ to $\hat{s}_{f+1}$, there is a non-empty set of concrete transitions from $h^{-1}(\hat{s}_f)$ to $h^{-1}(\hat{s}_{f+1})$ (Definition 3.2.2). The set B consists of the concrete states in $\hat{s}_f$ that have a transition to a concrete state in $\hat{s}_{f+1}$.

2. $f = k$: (Figure 4.5) The counterexample is spurious because there is no concrete transition from a state in D to a concrete state in $h^{-1}(\hat{s}_l)$. Since there is an abstract transition from $\hat{s}_k$ to $\hat{s}_l$, there is a non-empty set of concrete transitions from $h^{-1}(\hat{s}_k)$ to $h^{-1}(\hat{s}_l)$ (Definition 3.2.2). The set B consists of the concrete states in $\hat{s}_k$ that have a transition to a concrete state in $\hat{s}_l$.

3. $f = k + 1$: (Figure 4.6) The counterexample is spurious because no concrete trace, that leads to a state in D, loops back on to itself. The formula for the bad states is the same as the formula for the deadend states, except that it refers to states at cycle l, instead of cycle k. Thus, the formula $\mathcal{B}_{\mathcal{L}}^{k+1}(s_k)$ for the bad states is obtained by renaming variables s_{k+1} in the formula $\mathcal{D}_{\mathcal{L}}^{k+1}$ for the deadend states to new variables s_w, and then renaming variables s_l to s_{k+1} (Equation 4.13). The set B consists of the concrete states in $\hat{s}_l$ that lie on a trace that agrees with the counterexample and ends in a deadend state.

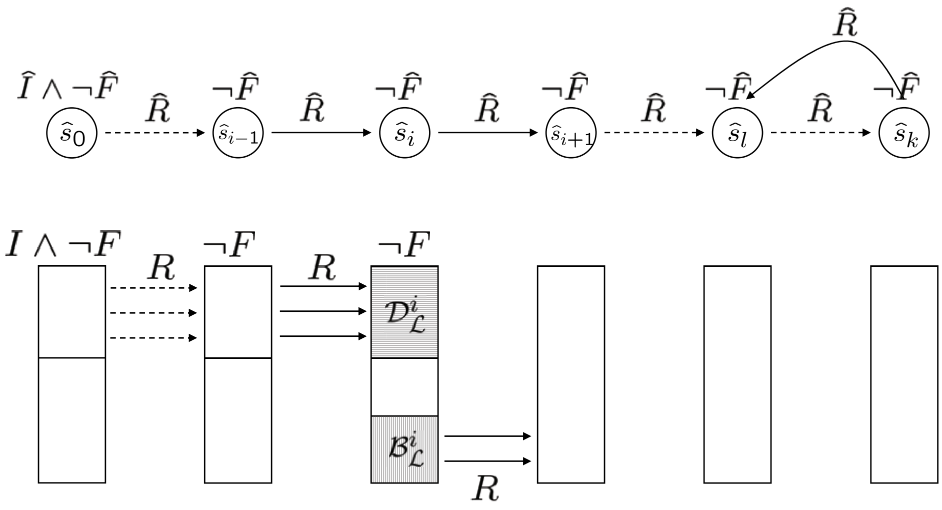

Figure 4.4: (Liveness property) The counterexample is spurious because there is no concrete transition from a deadend state to the next abstract state.

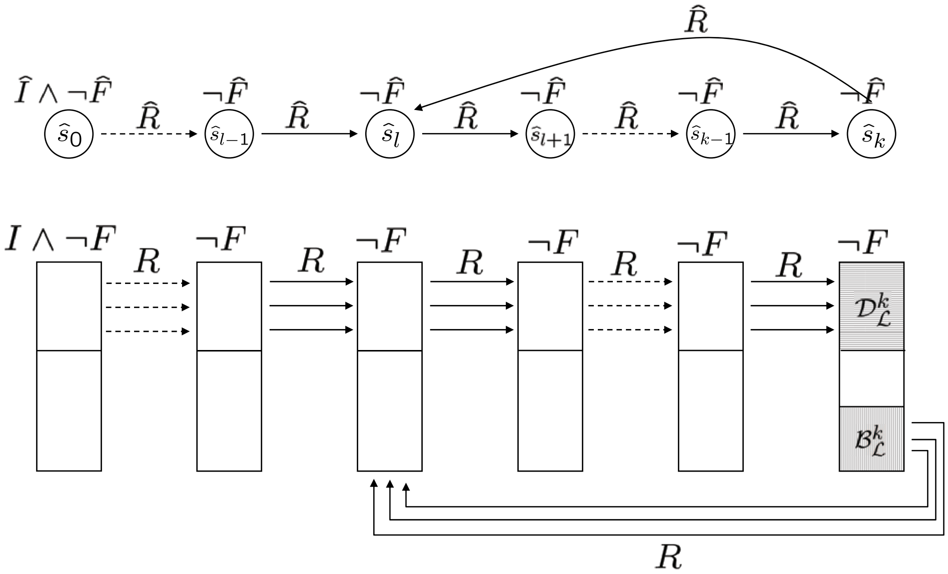

Figure 4.5: (Liveness property) The counterexample is spurious because there is no concrete transition corresponding to the abstract transition that loops back.

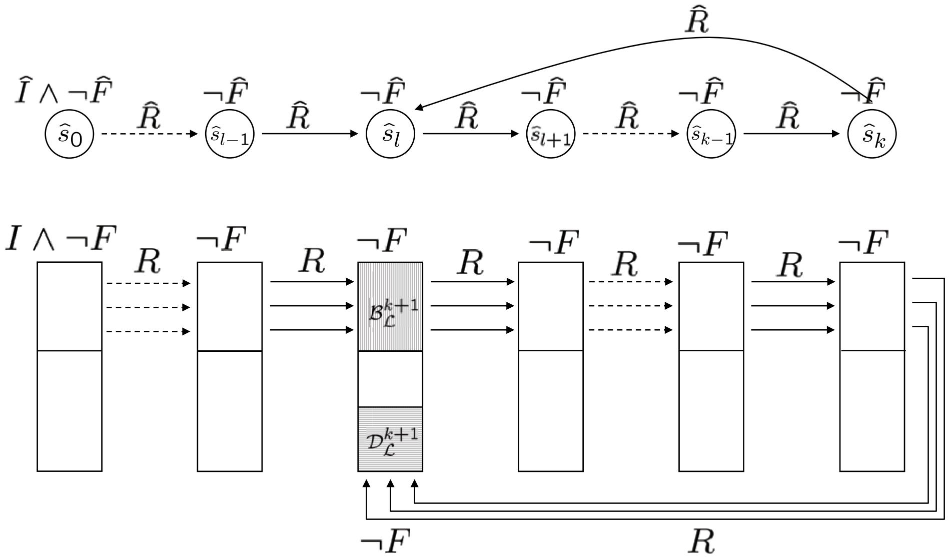

Figure 4.6: (Liveness property) The counterexample is spurious because no concrete trace, that leads to a deadend state, loops back on to itself.

4.5.3 Refinement as Separation

For both safety and liveness properties, the spurious counterexample exists because the deadend and bad states are mapped to the same abstract state. This motivates the following heuristic to eliminate the counterexample:

Our refinement strategy: (*Refinement as Separation*) The abstraction function h is refined to a new abstraction function h' such that

$$\forall d \in D, \forall b \in B \ (h'(d) \neq h'(b))$$

i.e. the new abstraction function puts the deadend and bad states into separate abstract states.

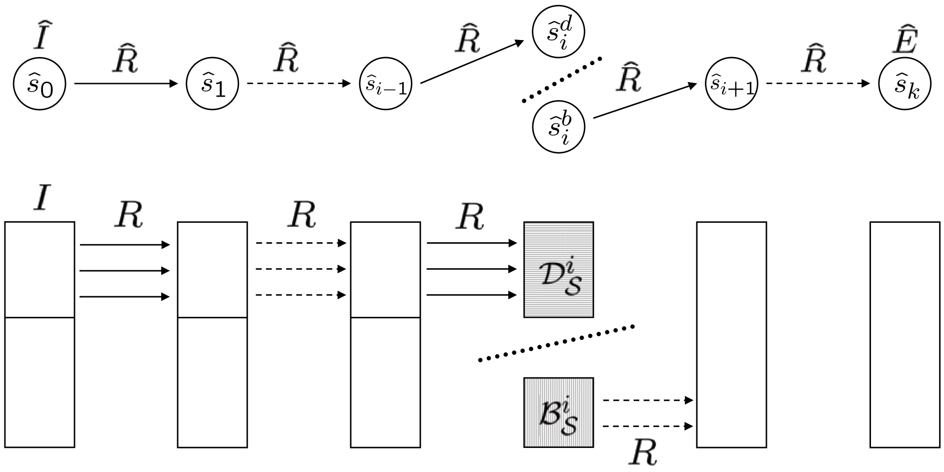

Figure 4.7: Effect of separating deadend and bad states for the counterexample in Figure 4.2.

4.5.4 Separation for Safety Properties

We illustrate the effect of separating deadend and bad states for the two scenarios in Section 4.5.1.

1. $0 \leq f < k$: (Figure 4.7) In the new abstract model, there is no transition from the abstract state that contains the deadend states to the next abstract state in the counterexample.

2. $f = k$: (Figure 4.8) In the new abstract model, the abstract state that contains the deadend states is not an error state.

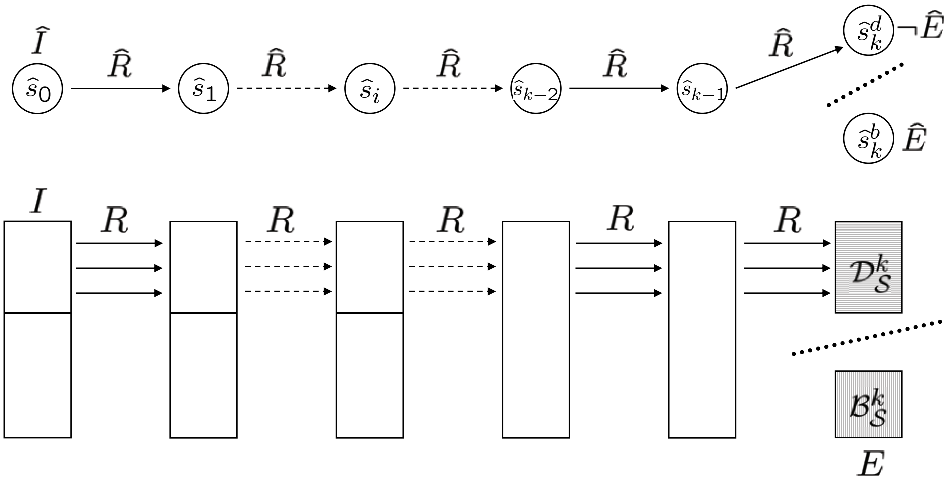

Figure 4.8: Effect of separating deadend and bad states for the counterexample in Figure 4.3.

4.5.5 Separation for Liveness Properties

We illustrate the effect of separating deadend and bad states for the three scenarios in Section 4.5.2.

1. $0 \leq f < k$: (Figure 4.9) In the new abstract model, there is no transition from the abstract state that contains the deadend states, to the next abstract state in the counterexample.

2. $0 \leq f < k+1$: (Figure 4.10) In the new abstract model, there is no transition from the abstract state that contains the deadend states, to the abstract state corresponding to the start of the loop in the abstract counterexample.

3. $f = k+1$ (Figure 4.11) In the new abstract model, there is no transition that loops back from the last abstract state in the counterexample.

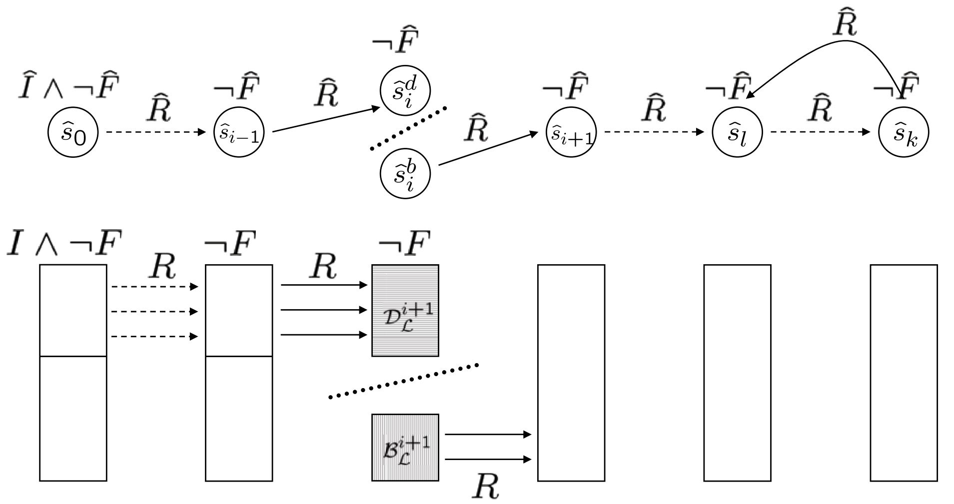

Figure 4.9: Effect of separating deadend and bad states for the counterexample in Figure 4.4.

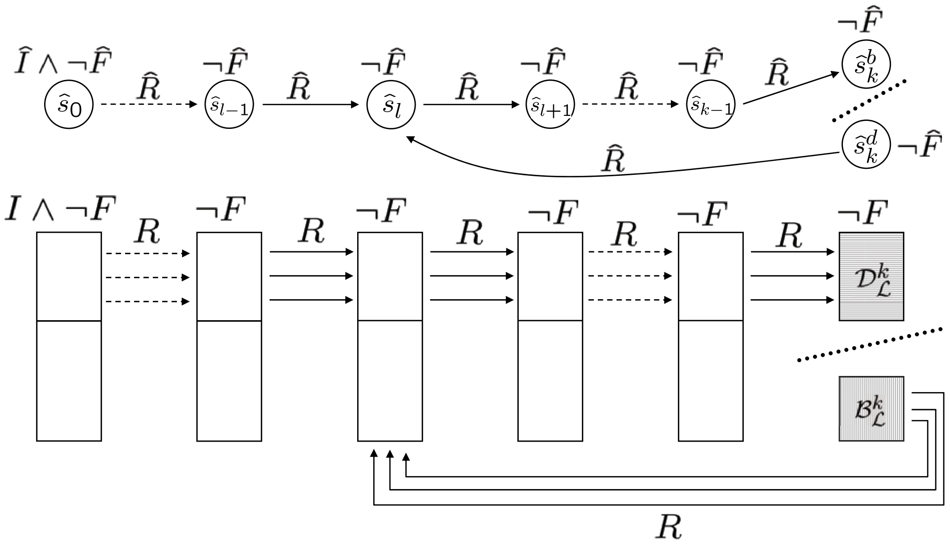

Figure 4.10: Effect of separating deadend and bad states for the counterexample in Figure 4.5.

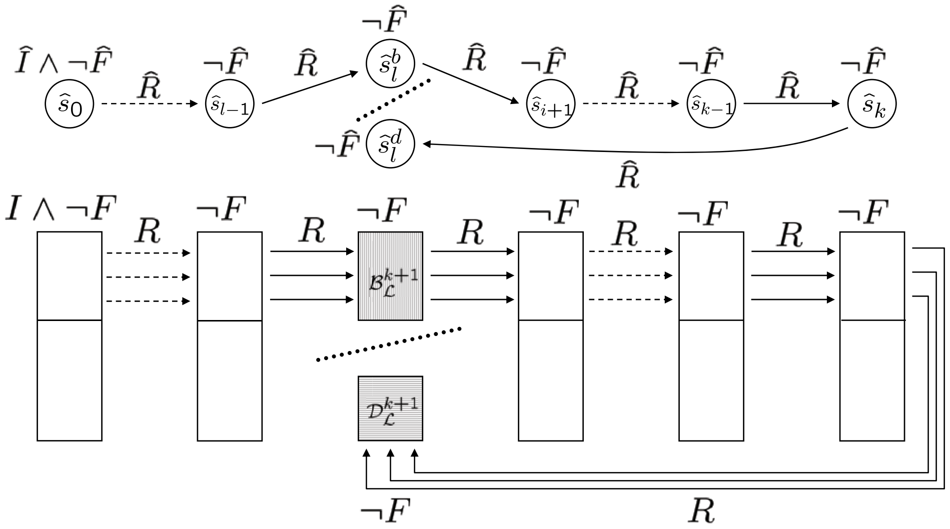

Figure 4.11: Effect of separating deadend and bad states for the counterexample in Figure 4.6.

4.5.6 The State Separation Problem

In the following definition, let $S = \{s_1, \dots s_m\}$ and $T = \{t_1, \dots t_n\}$ be two sets of states, where each state is an assignment to a set of variables W.

Definition 4.5.1. (*Separation of states with sets of variables*) A set of variables $U = \{u_1, \dots u_k\}$, $U \subseteq W$ *separates* S from T if for each pair of states (s_i, t_j), $s_i \in S$, $t_j \in T$, there exists a variable $u_r \in U$ such that $s_i(u_r) \neq t_j(u_r)$.

Definition 4.5.2. (*The state separation problem*) Given two sets of states S and T as defined above, find the smallest set of variables $U = \{u_1, \dots u_k\}$, $U \subseteq W$ that separates S from T. The set U is a *separating set* for S and T.

Let $H \in \mathcal{I}$ be a set of variables that separates D from B. The refinement is obtained by adding H to $\mathcal{V}$. The requirement that H is the smallest set is not crucial for the correctness of our approach, rather it is a matter of efficiency. Smaller sets of visible variables make it easier to model check the abstract system, but can also be harder to find. In fact, it has been shown that computing the minimal separating set is NP-hard [Clarke *et al.*, 2000].

Lemma 4.5.3. *Let H be a set of variables that separate the set of deadend states D from the set of bad states B. Let the abstraction function h' correspond to the visible set $\mathcal{V}' = \mathcal{V} \cup H$, where $\mathcal{V}$ is the current set of visible variables. Then h' maps D and B on to different abstract states in the abstract model.*

Proof. Let $d \in D$ and $b \in B$. Since H separates D and B, there exists a $u \in H$ s.t. $d(u) \neq b(u)$. Thus, for some $u \in \mathcal{V}'$, $d(u) \neq b(u)$. By definition (Equation 3.2), $h'(d) = (d(u_1) \ldots d(u_k))$ and $h'(b) = (b(u_1) \ldots b(u_k))$, $u_i \in \mathcal{V}'$ for all $1 \leq i \leq k$. Thus, $h'(d) \neq h'(b)$. □

4.6 Refinement as Learning

The naive way of separating the set of deadend states D from the set of bad states B would be to generate and separate D and B, either explicitly or symbolically. Unfortunately, for systems of realistic size, this is usually not possible. For all but the simplest examples, the number of states in D and B is too large to enumerate explicitly. For systems with moderate complexity, these sets can be computed symbolically with OBDDs. Experience shows, however, that there are many systems for which this is not possible [Clarke *et al.*, 2000]. Moreover, even if it was possible to generate D and B, it would still be computationally expensive to identify the separating variables.

Instead, we generate *samples* from D and B and *learn* the separating variables for the entire sets from these samples. In most cases, a small subset of D (denoted by S_D) and a small subset of B (denoted by S_B) is sufficient to infer the separating set, i.e. the separating set for S_D and S_B is also a separating set for D and B.

Our approach can be formulated as an inductive learning algorithm as follows: An abstraction function $h : S \to \hat{S}$ classifies states $s \in S$ into classifications $\hat{s} \in \hat{S}$. The goal of our inductive learner is to identify an abstraction function $h^{\mathcal{V}'}$ that is a refinement of the abstraction function $h^{\mathcal{V}}$ in the previous loop iteration, such that $h^{\mathcal{V}'}$ classifies the deadend states and the bad states into separate abstract states. The experience available to the learner is a set of deadend state samples S_D, and a set of bad state samples S_B. The learner searches for the function h' in the set $F = \{\ h^{\mathcal{V}'} \mid \mathcal{V} \subset \mathcal{V}' \subseteq V\ \}$ (Figure 4.12).

4.7 Generating Samples

We experimented with both *random sampling* and *equivalence queries* to generate the sample sets (Section 2.2). In the rest of the thesis, when the type of the property is not relevant, we will use the notation $\mathcal{D}$ to refer to both $\mathcal{D}^f_{\mathcal{S}}$ and $\mathcal{D}^f_{\mathcal{L}}$. Similarly, we will use $\mathcal{B}$ to refer to both $\mathcal{B}^f_{\mathcal{S}}$ and $\mathcal{B}^f_{\mathcal{L}}$.

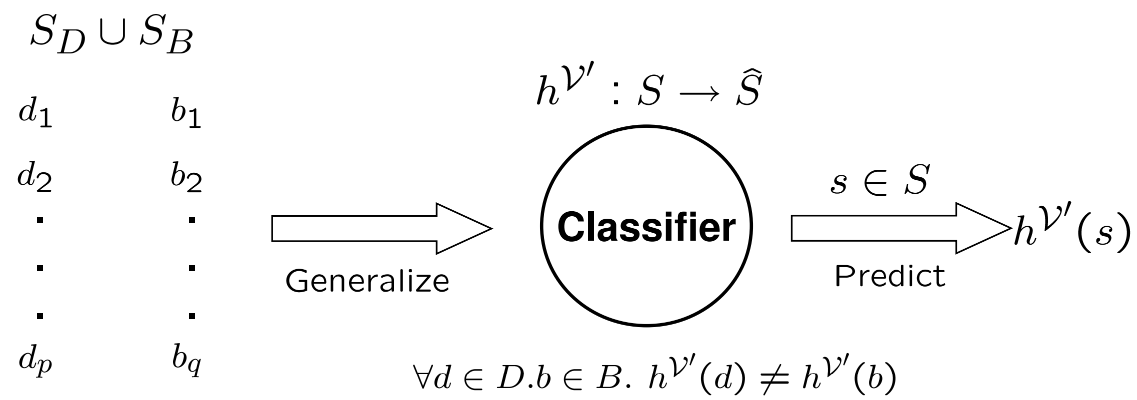

Figure 4.12: Abstraction-Refinement as Inductive Learning (see Figure 2.1).

4.7.1 Random Sampling

The sample sets are obtained by generating multiple satisfying assignments to $\mathcal{D}$ and $\mathcal{B}$. We used an incremental SAT-solver for generating these satisfying assignments. Every time a satisfying assignment is obtained, a clause that eliminates the assignment is added to the formula, and the incremental solver is asked to find an assignment to the new formula. The process is repeated until a specified number of samples have been generated; or until the formula becomes unsatisfiable, which happens when all the satisfying assignments have been enumerated. The number of samples to be generated is provided by the user. With random sampling, separating S_D and S_B does not guarantee that D and B are separated. Thus, this approach might not eliminate the counterexample in a single refinement step. However, the overall algorithm is complete because the counterexample is eventually eliminated in subsequent iterations of the CEGAR loop.

4.7.2 Sampling with Equivalence Queries

In the equivalence query model (Section 2.2), the learner has access to an oracle that takes as input a classifying function g and returns *yes* if g is the target function, otherwise it returns an object that is classified incorrectly by g. The learner generates a new sample by querying the oracle with a classifying function that it infers from the current set of samples. The object that is returned as an answer to this query contains more information than a randomly generated sample, since a randomly generated sample might be classified correctly by the classifying function computed from the current set of samples, and therefore might be redundant.

Sampling of D and B does not have to be arbitrary. Using equivalence queries, it is possible to direct the search to samples that contain more information than others. In the following section, we present an iterative algorithm that uses equivalence queries to sample D and B.

4.7.3 The ***Sample-and-Separate*** algorithm

Algorithm 4.1 describes *Sample-and-Separate* (SAMPLESEP), for sampling with equivalence queries. Given two sets of states S_1 and S_2, let $\delta(S_1, S_2)$ denote the separating set for S_1 and S_2. Let *SepS* denote the separating set that SAMPLESEP is working with at the current iteration. Initially, *SepS* is empty. SAMPLESEP iteratively adds or replaces elements in *SepS* until it becomes a separating set for D and B. In each iteration, the algorithm finds samples that are not separable by *SepS* that was computed in the previous iteration. Computing a new pair of deadend and bad states

Algorithm 4.1 Sampling with Equivalence Queries

SampleSep($\mathcal{D}, \mathcal{B}$)

1: $SepS = \emptyset$
2: $i = 0$
3: **while** (1) **do**
4:     **if** ($\Phi(\mathcal{D}, \mathcal{B}, SepS)$ is satisfiable) **then**
5:         Let d_i correspond to the assignment to the v_i variables
6:         Let b_i correspond to the assignment to the v'_i variables
7:         $SepS = \delta(\bigcup_{j=0}^{i}\{d_j\},\ \bigcup_{j=0}^{i}\{b_j\})$
8:         $i = i + 1$
9:     **else**
10:         **return** $SepS$

that are not separable by $SepS$ is done by solving $\Phi(\mathcal{D}, \mathcal{B}, SepS)$, as defined below:

$$\Phi(\mathcal{D}, \mathcal{B}, SepS) \; = \; \mathcal{D} \;\wedge\; \mathcal{B}' \;\wedge \bigwedge_{v_i \in SepS} v_i = v'_i \tag{4.19}$$

The prime symbol over $\mathcal{B}$ denotes the fact that we replace each variable v_i in $\mathcal{B}$ with a new variable v'_i (note that otherwise, by definition, the conjunction of $\mathcal{D}$ with $\mathcal{B}$ is unsatisfiable). The right-most conjunct in the above formula guarantees that the new samples of deadend and bad states are not separable by the current separating set.

In each iteration, SampleSep solves Formula 4.19 (line 4). In the learning with equivalence queries model, this corresponds to an equivalence query to the oracle. If the formula is satisfiable, it implies that *SepS* is different from the target separating set for D and B. In this case, SampleSep derives the samples $d_i \in D$ and $b_i \in B$ from the satisfying assignment (these are simply the assignments to the variables in $\mathcal{D}$ and $\mathcal{B}'$, respectively), which by definition are not separable by the current separating set *SepS* (lines 5,6). It then re-computes *SepS* for the union of the samples that were computed up to the current iteration (line 7). If Formula 4.19 is unsatisfiable, it implies that *SepS* is the same as the target separating set for D and B. By repeating the loop until Formula 4.19 becomes unsatisfiable, it guarantees that the resulting separating set separates D from B.

In each iteration, SampleSep finds a single solution to $\Phi(\mathcal{D}, \mathcal{B}, SepS)$, and hence a single pair of states d_i and b_i. However, the number of samples in each iteration can be larger. Larger number of samples may reduce the number of iterations, but also require more time to derive and separate. The optimal number of new samples in each iteration depends on various factors, like the efficiency of the SAT-solver, the technique used to compute the separating set, and the examined model. Our implementation lets the user control this process by adjusting two parameters: the number of samples generated in each iteration, and the maximum number of iterations.

4.8 Computing the Separating Set

We experimented with two techniques for computing the separating set from the sample sets S_D and S_B. The first technique uses 0-1 Integer Linear Programming

$$\text{Min } \sum_{i=1}^{|\mathcal{I}|} v_i$$

$$\text{subject to:} \quad (\forall s \in S_D)\ (\forall t \in S_B) \sum_{\substack{1 \leq i \leq |\mathcal{I}|, \\ s(v_i) \neq t(v_i)}} v_i \geq 1$$

Figure 4.13: State Separation with Integer Linear Programming.

(ILP), while the second technique is based on Decision Tree Learning (DTL). The next two sections describe these techniques.

4.8.1 Separation using Integer Linear Programming

A formulation of the problem of separating S_D from S_B as a 0-1 Integer Linear Programming problem is depicted in Figure 4.13. The value of each Boolean variable $v_1 ... v_{|\mathcal{I}|}$ in the ILP problem is interpreted as: $v_i = 1$ if and only if v_i is in the separating set. Every constraint corresponds to a pair of states (s_i, t_j), stating that at least one of the variables that separates (distinguishes) between the two states should be selected. Thus, there are $|S_D| \times |S_B|$ constraints in the ILP formulation.

Example 4.8.1. Consider the sample sets $S_D = \{s_1, s_2\}$ and $S_B = \{t_1, t_2\}$, where:

$$s_1 = (0, 1, 0, 1) \qquad t_1 = (1, 1, 1, 1)$$
$$s_2 = (1, 1, 1, 0) \qquad t_2 = (0, 0, 0, 1)$$

Assume that all the variables are invisible. Let v_i correspond to the i-th component of a state. Then the corresponding ILP for computing the separating set for S_D and S_B is:

Min $\sum_{i=1}^{4} v_i$

subject to:

$$
\begin{array}{lll}
v_1 + v_3 & \geq 1 & \text{// Separating } s_1 \text{ from } t_1 \\
v_2 & \geq 1 & \text{// Separating } s_1 \text{ from } t_2 \\
v_4 & \geq 1 & \text{// Separating } s_2 \text{ from } t_1 \\
v_1 + v_2 + v_3 + v_4 & \geq 1 & \text{// Separating } s_2 \text{ from } t_2
\end{array}
$$

The optimal value of the objective function in this case is 3, corresponding to one of the two optimal solutions $\{v_1, v_2, v_4\}$ and $\{v_3, v_2, v_4\}$.

4.8.2 Separation using Decision Tree Learning

The ILP-based separation algorithm outputs the minimal separating set. Since ILP is NP-complete, we also experimented with a polynomial approximation based on Decision Tree Learning. This technique is polynomial both in the number of variables and the number of samples, but does not necessarily give optimal results.

Learning with decision trees is one of the most widely used and practical methods for approximating discrete-valued functions. A DTL algorithm inputs a set of examples. An example is described by a set of attributes and the corresponding classification. The algorithm generates a decision tree that classifies the examples. Each internal node in the tree specifies a test on some attribute, and each branch descending from that node corresponds to one of the possible values for that attribute. Each leaf in the tree corresponds to a classification.

Data is classified by a decision tree by starting at the root node of the decision tree, testing the attribute specified by this node, and then moving down the tree branch corresponding to the value of the attribute. The process is repeated for the sub-tree rooted at the branch until one of the leafs is reached, which is labeled with the classification.

The problem of separating S_D from S_B can be formulated as a DTL problem as follows:

- The attributes correspond to the invisible variables.
- The classifications are $+1$ and -1, corresponding to S_D and S_B, respectively.
- The examples are S_D labeled $+1$, and S_B labeled -1.

We generate a decision tree for this DTL problem. From this tree we extract all the variables present at the internal nodes. These variables constitute the separating set.

Lemma 4.8.2. *The formulation based on Decision Tree Learning outputs a separating set for S_D and S_B.*

Proof. Let $d \in S_D$ and $b \in S_B$. The decision tree will classify d as $+1$ and b as -1. So, there exists a node n in the decision tree, labeled with a variable v, such that $d(v) \neq b(v)$. By construction, v lies in the output set. Thus, by Definition 4.5.1, the DTL formulation outputs the separating set for S_D and S_B. □

Example 4.8.3. Going back to Example 4.8.1, the corresponding DTL problem has 4 attributes $\{v_1, v_2, v_3, v_4\}$ and as always, two classifications $\{+1, -1\}$. The set of examples contains the following elements:

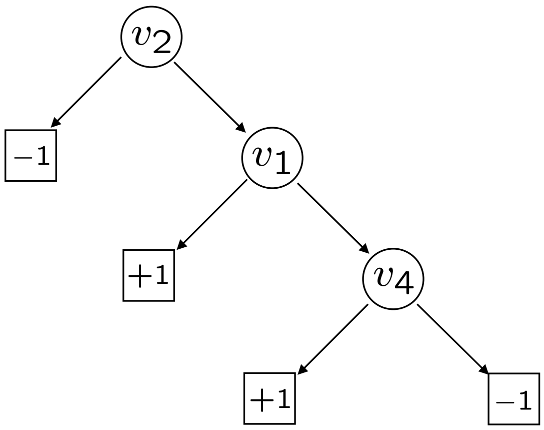

Figure 4.14: Decision Tree for Example 4.8.3.

$$((0,1,0,1),+1) \qquad ((1,1,1,1),-1)$$
$$((1,1,1,0),+1) \qquad ((0,0,0,1),-1)$$

The tree appearing in figure 4.14 classifies these examples. It corresponds to the separating set $\{v_1, v_2, v_4\}$.

A number of algorithms have been developed for learning decision trees, e.g. ID3 [Quinlan, 1986], C4.5 [Quinlan, 1993]. All these algorithms essentially perform a simple top-down greedy search through the space of possible decision trees. We implemented a simplified version of the ID3 algorithm [Mitchell, 1997], which is described in Algorithm 4.2.

DecTree is a recursive algorithm that takes as input a set of examples E and a set of attributes A. It returns a decision tree whose nodes are labeled with the attributes in A, and which classifies the examples in E. At each recursion, DecTree

Algorithm 4.2 Decision Tree Learning

DecTree(E, A)

1: Create a root node R for the tree

2: If all examples in E are classified the same, return R with this classification

3: Let $a = BestAttribute(E, A)$. Label R with attribute a

4: For $i \in \{0, 1\}$, let E_i be the subset of E having value i for a

5: For $i \in \{0, 1\}$, add an i branch to R pointing to sub-tree generated by $Dectree(E_i, A - \{a\})$

6: **return** R

calls the *BestAttribute* function to pick an attribute to test at the root (line 3). In order to make that choice, we need a measure of the quality of an attribute. We start with defining a quantity called *entropy* [Mitchell, 1997], which is a commonly used notion in information theory.

Definition 4.8.4. Given a set S containing $n_\oplus$ positive examples and $n_\ominus$ negative examples, the *entropy* of S is given by:

$$Entropy(S) = -p_\oplus log_2 p_\oplus - p_\ominus log_2 p_\ominus$$

where $p_\oplus = (n_\oplus)/(n_\oplus + n_\ominus)$ and $p_\ominus = (n_\ominus)/(n_\oplus + n_\ominus)$.

Intuitively, entropy characterizes the variety in a set of examples. The maximum value for entropy is 1, which corresponds to a collection that has an equal number of positive and negative examples. The minimum value of entropy is 0, which corresponds to a collection with only positive or only negative examples. We can now define the quality of an attribute a by the reduction in entropy on partitioning the examples using a. This measure, called the *information gain* [Mitchell, 1997] is defined as follows:

Definition 4.8.5. The *information gain* of an attribute a with respect to a set of samples E is calculated as follows:

$$Gain(E,a) = Entropy(E) - (|E_0|/|E|) \cdot Entropy(E_0) - (|E_1|/|E|) \cdot Entropy(E_1)$$

where E_0 and E_1 are the subsets of examples having the value 0 and 1, respectively, for attribute a.

The $BestAttribute(E, A)$ function returns the attribute $a \in A$ that has the highest $Gain(E, A)$. Its complexity is $O(|E||A|)$.

Example 4.8.6. We illustrate the working of our algorithm with an example. Continuing with Example 4.8.1, we calculate the gains for the attributes at the top node of the decision tree.

$$Entropy(E) = -(2/4)log_2(2/4) - (2/4)log_2(2/4) = 1.00$$

$$Gain(E, v_1) = 1 - (2/4) \cdot Entropy(E_{v_1=0}) - (2/4) \cdot Entropy(_{v_1=1}) = 0.00$$

$$Gain(E, v_2) = 1 - (1/4) \cdot Entropy(E_{v_2=0}) - (3/4) \cdot Entropy(_{v_2=1}) = 0.31$$

$$Gain(E, v_3) = 1 - (2/4) \cdot Entropy(E_{v_3=0}) - (2/4) \cdot Entropy(_{v_3=1}) = 0.00$$

$$Gain(E, v_4) = 1 - (1/4) \cdot Entropy(E_{v_4=0}) - (3/4) \cdot Entropy(_{v_4=1}) = 0.31$$

The DecTree algorithm will pick v_2 or v_4 to label the root. Figure 4.14 shows a possible output of DecTree for this example set.

4.9 Changing the objective

The criterion of minimum number of variables that separate a given set of samples is not necessarily the optimal one for faster model checking. Through experiments we discovered that minimizing the number of inputs in the abstract model is far better for reducing the complexity of model checking (one of the expensive stages in model checking is removing the quantifiers over all inputs).

4.9.1 Minimizing Inputs with ILP

Let In denote the set of primary inputs. In order to find the set of separating variables that minimizes the number of inputs in the resulting abstract model, we derive a mapping $\mathcal{I} \rightarrow (2^{\mathcal{I} \cup In})$ that maps each invisible variable to the set of variables in its (first-layer) fan-in that are not yet in the model. Let $FanIn(v)$ denote this set for a variable v, and let $\mathcal{F} = \bigcup_{j=1}^{|\mathcal{I}|} FanIn(v_j)$. With ILP, we can now encode each variable $v \in \mathcal{I}$ with a new Boolean variable and add a constraint stating that if v is true, then so are all the variables in $FanIn(v)$. Minimizing over the sum of inputs gives us the desired result. Figure 4.15 gives a 0-1 ILP formulation of this problem.

Example 4.9.1. Continuing with Example 4.8.1, suppose that we derive the following mapping of invisible variables to variables in their fanin:

$$v_1 \rightarrow \{i_1, i_3\} \qquad v_2 \rightarrow \{i_1, i_5\} \qquad v_3 \rightarrow \{i_2\} \qquad v_4 \rightarrow \{i_3, i_4\}$$

The corresponding ILP is (here we write the new set of constraints as propositional formulas rather than inequalities, for clarity):

Min $\sum_{i=1}^{|\mathcal{F}|} v_i$

Subject to:

1. $\forall s \in S_D.\ \forall t \in S_B.\ \sum_{\substack{1 \le i \le |\mathcal{I}|, \\ s(v_i) \neq t(v_i)}} v_i \ge 1$

2. $\forall 1 \le i \le |\mathcal{I}|.\ \forall v_j \in FanIn(v_i).\ v_j - v_i \ge 0 \quad //$ Same as $v_i \rightarrow v_j$

Figure 4.15: A 0-1 ILP formulation of the state separation problem, where the objective is to minimize the number of inputs in the abstract model.

Min $\sum_{i=1}^{5} i_i$

subject to:

...... // same constraints as in Example 4.8.1

$v_1 \rightarrow i_1 \wedge i_3$

$v_2 \rightarrow i_1 \wedge i_5$

$v_3 \rightarrow i_2$

$v_4 \rightarrow i_3 \wedge i_4$

There are three possible satisfying assignments to the constraints that appeared in Example 4.8.1: $\{v_1, v_2, v_4\}, \{v_3, v_2, v_4\}$ and $\{v_1, v_2, v_3, v_4\}$. Only the first option minimizes the number of inputs to four.

4.9.2 Minimizing Inputs with DTL

For DTL-based separation, we minimize the number of inputs by assigning a higher cost to the attributes corresponding to state variables that introduce more inputs into the abstract model. Our DecTree algorithm can be modified to take into account attribute costs by introducing a cost measure into the *BestAttribute* function. We replaced the information gain attribute selection measure by the following measure [Tan and Schlimmer, 1990]:

$$\frac{Gain^2(E,a)}{Cost(E,a)}$$

Similar to unweighted DTL, such cost-sensitive measures do not guarantee finding an optimal weighted decision tree. They only bias the search in favor of attributes that have a lower cost.

In all our experiments, minimizing over the number of inputs turned out to be more efficient than simply minimizing the number of state variables. In some cases, it enabled us to solve instances that we could not solve with the previous method in the given time and memory bounds.

4.10 Generating Good Samples

Through experiments, we found that the number of sampling iterations in Sample-Sep can become a bottleneck. This problem is at least partially a consequent of the arbitrariness of selecting an optimal solution when there are multiple equally good possibilities. For example, in the biggest circuit we experimented with (which has 5000 state variables), a sample pair of deadend and bad states (d, b) would differ typically in more than a thousand variables. Our optimization engine selects one of these variables arbitrarily, say v_1. Then a new pair of states (d', b') is sampled, and again, more than a thousand options exist to separate the sets $(\{d, d'\}, \{b, b'\})$ with a single variable. If any one of these options was selected in the first iteration rather than v_1, it would have ruled out the sample pair (d', b').

This problem can be formally explained as follows. The SampleSep algorithm computes a separating set that is a subset of the set of invisible variables $\mathcal{I}$. Thus, the number of potential candidates that SampleSep has to consider is $2^{|\mathcal{I}|}$. Given a sample pair (d, b), the variables $W = \{w_1, \ldots w_k\}$ on which d and b differ are called the *distinguishing variables* for (d, b). Whenever SampleSep generates a sample pair (d, b), it infers that at least one variable from the set of distinguishing variables W for (d, b) should be present in the final separating set. Therefore, this sample pair allows SampleSep to eliminate (from the set of potential candidates) all the subsets of $\mathcal{I}$ that do not contain a variable from W. There are $2^{(|\mathcal{I}|-|W|)}$ such subsets. A sample pair with fewer distinguishing variables (corresponding to a small value for $|W|$) is a *good* sample pair because it would eliminate a large portion of the search space for SampleSep, and would lead to faster convergence of the algorithm.

In each iteration of SampleSep, selecting the sample pair that has the minimum number of distinguishing variables corresponds to finding a satisfying assignment to $\Phi(\mathcal{D}, \mathcal{B}, SepS)$ (Equation 4.19) that minimizes the number of pairs (v_i, v_i') that are evaluated differently. We solve this boolean optimization problem with Pseudo-

Boolean Solver (PBS) [Aloul *et al.*, 2002].

A Psuedo-Boolean Constraint (PBC) [Aloul *et al.*, 2002] is a linear constraint over boolean variables:

$$\sum c_i x_i \leq n \qquad c_i, n \in Z, x_i \in \{0, 1\}$$

PBS extends the standard Davis-Putnam procedure to solve a formula consisting of a conjunction of propositional clauses and PBCs. It uses special Boolean Constraint Propagation (BCP) rules for handing PBCs. This approach is more efficient compared to expanding the PBCs into a collection of propositional clauses, because such an expansion is exponential. PBS solves the boolean optimization problem by adding a PBC corresponding to the objective function and progressively increasing its value till the resulting formula becomes unsatisfiable.

In order to generate sample pairs with the smallest number of distinguishing variables, the input to PBS is the following propositional formula:

$$\Phi(\mathcal{D}, \mathcal{B}, SepS) \;\wedge\; \bigwedge_{v_i \in \mathcal{I}} (d_i = (v_i = v_i')) \tag{4.20}$$

where $d_i's$ are fresh variables. Intuitively, a satisfying assignment to Equation 4.20 assigns a value 1 to d_i if and only if the variable v_i has the same value in the corresponding deadend and bad state sample, which implies that v_i is not a separating variable for these states. PBS is asked to maximize $\sum d_i$. We tuned PBS for our instances by forcing it to first split on variables in the objective function (the d_i

variables), and to first try the value 1 for these variables since it is a maximization problem. This strategy turned out to be far superior to standard dynamic orderings.

Due to the generation of good samples, in some of the big examples, we witnessed a decrease of two orders of magnitude in the number of iterations that are required for convergence of the sampling algorithm.

4.11 Example

We will go over each step of our CEGAR implementation, using the model $\mathcal{M}$ described in Section 3.4 as an example. We will illustrate the configuration that uses SampleSep for generating the samples, ILP for computing the separating set, and PBS for generating good samples. The deadend and bad states in the sample sets shown below are assignments to the state variables $\{x, y, z, u, c_0.v, c_1.v, c_2.v\}$. In the sample sets, assignments to variables in the separating set are shown in a bold font.

- *Abstract (Iteration* 1*) :* The initial abstract model $\mathcal{M}_1$, corresponding to $\mathcal{V} = \{\}$, is shown in Figure 4.16.
- *Model Check (Iteration* 1*) :* Model checking $\mathcal{M}_1$ produces the counterexample shown in Figure 4.17.
- *Spurious*? *(Iteration* 1*) :* The formula ϕ_{C_S} (Equation 4.2) for the counterexample in Figure 4.17 is unsatisfiable. Thus, the counterexample is spurious.
- *Refine (Iteration* 1*) :* The counterexample is of length $k = 0$. Hence, the

```
MODULE main
IVAR
        i, j, x, y, z, u, c0, c1, c2 : boolean;
SPEC AG u
```

Figure 4.16: The initial abstract model $\mathcal{M}_1$.

State: ()

()

$\neg u$

Figure 4.17: Counterexample on model $\mathcal{M}_1$ (Figure 4.16).

failure index f is 0. The final sample sets and the separating set generated by SampleSep are:

$$S_D = \{(1, 0, 0, \mathbf{1}, 0, 0, 0)\}$$

$$S_B = \{(1, 0, 0, \mathbf{0}, 0, 0, 0)\}$$

$$SepS = \{u\}$$

- *Abstract (Iteration 2) :* The abstract model $\mathcal{M}_2$ for iteration 2, corresponding to $\mathcal{V} = \{u\}$, is shown in Figure 4.18.
- *Model Check (Iteration 2) :* Model checking $\mathcal{M}_2$ produces the counterexample shown in Figure 4.19.

```
MODULE main
VAR
        u : boolean;
IVAR
        i, j, x, y, z, c0, c1, c2 : boolean;
ASSIGN
        init(u) := 1;
        next(u) := case
                        c0 & c1 & c2 : x | y | z;
                        1 : u;
                      esac;
SPEC AGu
```

Figure 4.18: The abstract model $\mathcal{M}_2$.

- *Spurious? (Iteration* 2*) :* The formula $\phi_{C_{\mathcal{S}}}$ (Equation 4.2) for the counterexample in Figure 4.19 is unsatisfiable. Thus, the counterexample is spurious.

- *Refine (Iteration* 2*) :* The largest index i for which the formula $\mathcal{D}^i_{\mathcal{S}}$ (Equation 4.8) is satisfiable is 0. Hence, the failure index f is 0. The final sample sets and the separating set generated by SampleSep are:

$$S_D = \{(1,0,0,1,\mathbf{0},0,0),(0,1,0,1,\mathbf{0},0,0)\}$$

$$S_B = \{(0,0,0,1,\mathbf{1},1,1)\}$$

$$SepS = \{c_0.v\}$$

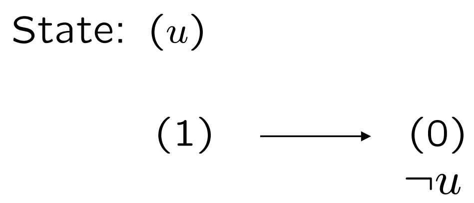

Figure 4.19: Counterexample on model $\mathcal{M}_2$ (Figure 4.18).

- *Abstract (Iteration* 3*) :* The abstract model $\mathcal{M}_3$ for iteration 3, corresponding to $\mathcal{V} = \{u, c_0.v\}$, is shown in Figure 4.20.

- *Model Check (Iteration* 3*) :* Model checking $\mathcal{M}_3$ produces the counterexample shown in Figure 4.21.

- *Spurious? (Iteration* 3*) :* The formula $\phi_{C_\mathcal{S}}$ (Equation 4.2) for the counterexample in Figure 4.21 is unsatisfiable. Thus, the counterexample is spurious.

- *Refine (Iteration* 3*) :* The largest index i for which the formula $\mathcal{D}^i_\mathcal{S}$ (Equation 4.8) is satisfiable is 1. Hence, the failure index f is 1. The final sample sets and the separating set generated by SampleSep are:

$$S_D = \{(0, 1, \mathbf{1}, 1, 1, 0, 0), (1, 0, \mathbf{1}, 1, 1, 0, 0)\}$$

$$S_B = \{(0, 0, \mathbf{0}, 1, 1, 1, 1)\}$$

$$SepS = \{z\}$$

- *Abstract (Iteration* 4*) :* The abstract model $\mathcal{M}_4$ for iteration 4, corresponding to $\mathcal{V} = \{z, u, c_0.v\}$, is shown in Figure 4.22.

```
MODULE main
VAR
      u : boolean;
      c0 : counter_cell(1);
IVAR
      i, j, x, y, z, c1, c2 : boolean;
ASSIGN
      init(u) := 1;
      next(u) := case
                    c0.v & c1 & c2 : x | y | z;
                    1 : u;
                 esac;
SPEC AGu

MODULE counter_cell(cin)
VAR
      v : boolean;
ASSIGN
      init(v) := 0;
      next(v) := v + cin mod 2;
DEFINE
      cout := v & cin;
```

Figure 4.20: The abstract model $\mathcal{M}_3$.

State: $(u, c_0.v)$

$$(1,0) \longrightarrow (1,1) \longrightarrow \underset{\neg u}{(0,0)}$$

Figure 4.21: Counterexample on model $\mathcal{M}_3$ (Figure 4.20).

- *Model Check (Iteration* 4*) :* The property holds on $\mathcal{M}_4$, and therefore the property holds on $\mathcal{M}$. This terminates the CEGAR loop.

4.12 Related Work

The closest work to our approach is described in Yuan Lu's thesis [Lu, 2000] and is more briefly summarized in [Clarke *et al.*, 2000]. Like our approach, they also use an automatic, iterative abstraction-refinement procedure that is guided by the counterexample, and they also try to eliminate the counterexample by solving the state-separation problem. But there are three main differences between the two methods. First, their abstraction is based on replacing predicates of the program with new input variables, while our abstraction is performed by making some of the variables invisible (thus, we hide the entire logic that defines these variables). The advantage of our approach is that computing the minimal abstract model becomes easy. Secondly, checking whether the counterexample is real or spurious was performed in their work symbolically, using OBDDs. We do this stage with a SAT-solver, which is extremely efficient for this particular task (due to the large number of solutions to the SAT instance). Thirdly, they derive the refinement symbolically. Since finding the coarsest refinement is NP-hard, they present a polynomial procedure that in general computes a sub-optimal solution. For some well defined cases the same procedure computes the optimal refinement. We, on the other hand, tackle this complexity by considering only samples of the states sets, which we compute explicitly.

```
MODULE main
VAR
    z, u : boolean;
    c0 : counter_cell(1);
IVAR
    i, j, x, y, c1, c2 : boolean;
ASSIGN
    init(z) := 0;
    next(z) := !z;

    init(u) := 1;
    next(u) := case
                   c0.v & c1 & c2 : x | y | z;
                   1 : u;
               esac;
SPEC AG u

MODULE counter_cell(cin)
VAR
    v : boolean;
ASSIGN
    init(v) := 0;
    next(v) := v + cin mod 2;
DEFINE
    cout := v & cin;
```

Figure 4.22: The abstract model $\mathcal{M}_4$. This model proves the property.

The work of Das et al. [Das and Dill, 2001] should also be mentioned in this context, since it is very similar to [Clarke *et al.*, 2000], the main difference being the refinement algorithm: rather than computing the refinement by analyzing the abstract failure state, they combine a theorem prover with a greedy algorithm that finds a small set of previously abstracted predicates that eliminate the counterexample. They add this set of predicates as a new constraint to the abstract model.

Previous work on abstraction by making variables invisible (this technique was used under different names in the past) include the localization reduction of Kurshan [Kurshan, 1995] and others (see, for example [Balarin and Sangiovanni-Vincentelli, 1993; Lind-Nielsen and Andersen, 1999]). The localization reduction follows the typical abstraction-refinement iterative process. It starts by making all but the property variables invisible. When a spurious counterexample is identified, it refines the system by making more variables visible. The variables made visible are selected according to the variable dependency graph and information that is derived from the counterexample. The candidates in the next refinement step are those invisible variables that are adjacent to the currently visible variables on the variable dependency graph. Choosing among these variables is done by extracting information from the counterexample. Another relevant work is described by Wang et al. in [Wang *et al.*, 2001]. They use 3-valued simulation to simulate the counterexample on the concrete model and identify the invisible variables whose values in the concrete model conflict with the counterexample. Variables are chosen from this set of invisible variables by various ranking heuristics. For example, like localization, they prefer variables that are close to the currently visible variables in the variable dependency graph.

More recent research by Chauhan et al. [Chauhan *et al.*, 2002] follows a technique that is very similar to ours. Like our approach, they also look for the failing state with a SAT-solver. But rather than analyzing the failing state, they derive information from the SAT-solver that explains why the spurious counterexample cannot be simulated beyond this state on the concrete machine. More specifically, they build an unsatisfiability proof by joining conflict graphs in the SAT-solver, and make visible all the variables from the failing state that correspond to vertices in this graph. As a second step, they try to minimize this set by gradually making some of these variables invisible again, and check whether this makes the instance satisfiable. The success of the second phase depends on the (arbitrarily chosen) order in which they remove the variables.

This approach has both advantages and disadvantages when compared to ours. The main advantage is that their refinement step consists of solving one SAT instance and analyzing the proof of unsatisfiability of this instance. We, on the other hand, look for an optimal solution, and therefore solve an optimality problem that can potentially take more time. By doing so we hope to make the model checking step faster. In general there is less arbitrariness in our procedure compared to theirs. In practice it is hard to compare the two methods because of this arbitrariness. For example, it is possible that due to two equally good refinements that the two tools perform, the next counterexample that they need to analyze is different (the counterexamples that model checkers produce are chosen arbitrarily from an exponential number of options). This can drastically change the results of the overall procedure. For this reason it is possible that their tool occasionally finds smaller abstract models compared to ours. We will refer to this point further when describing our experimental results.

4.13 Experimental Results

We implemented our framework inside NuSMV [Cimatti *et al.*, 2000]. We used NuSMV as a front-end, for parsing SMV files and for generating abstractions. However, for actual model checking, we used Cadence SMV [McMillan], which implements techniques like cone-of-influence reduction, cut-points, etc. We implemented a variant of the ID3 algorithm (Algorithm 4.2) to generate decision trees. We used Chaff [Moskewicz *et al.*, 2001] as our SAT-solver. Some modifications were made to Chaff to efficiently generate multiple state samples in a single run. We used the mixed ILP tool LP-Solve [Berkelaar] to compute the separating set, and we used PBS [Aloul *et al.*, 2002] to generate good samples.

Our experiments were performed on the "IU" family of circuits, which are various localization abstractions of an interface control circuit from Synopsys, Incorporated. We also experimented with several other circuits from various other sources, as we report in Table 4.3. All experiments were performed on a 1.5GHz Dual Athlon machine with 3GB RAM and running Linux. No pre-computed variable ordering files were used in the experiments.

The results for the "IU" family are presented in Table 4.1 and Table 4.2. The two tables correspond to two different properties. We compared the following techniques:

Circuit	SMV	Rand, ILP			Rand, ILP			Eqv, DTL			Eqv, DTL, Inp		
	Time	Time	S	L	Time	S	L	Time	S	L	Time	S	L
*IU*30	0.7	0.1	0	1	0.1	0	1	**0.1**	0	1	0.14	0	1
*IU*35	0.6	0.1	0	1	0.1	0	1	**0.1**	0	1	**0.1**	0	1
*IU*40	1.2	6.3	3	4	0.9	5	6	0.6	2	3	**0.59**	1	3
*IU*45	37.5	6.1	3	4	1.1	5	6	**0.7**	2	3	0.85	1	3
*IU*50	23.3	19.7	13	14	9.8	13	14	24.0	4	17	**4.02**	2	9
*IU*55	-	-	-	-	2072	6	9	3.0	1	6	**0.37**	0	1
*IU*60	-	7.8	4	7	7.8	4	7	4.5	1	6	**0.48**	0	1
*IU*65	-	7.9	4	7	7.9	4	7	3.8	1	5	**0.51**	0	1
*IU*70	-	8.1	4	7	8.2	4	7	3.8	1	5	**0.47**	0	1
*IU*75	102.9	32.0	9	10	24.5	13	14	24.1	2	7	**0.32**	0	1
*IU*80	603.7	31.7	9	10	44.0	13	14	24.1	2	7	**0.37**	0	1
*IU*85	2832	33.1	9	10	44.6	13	14	25.2	2	7	**0.36**	0	1
*IU*90	-	33.0	9	10	44.6	13	14	25.4	2	7	**0.35**	0	1

Table 4.1: Model checking results for property 1.

1. 'SMV': Cadence SMV.

2. 'Rand, ILP': Random sampling, separation using LP-solve, 50 samples per refinement iteration, minimizing number of state variables.

3. 'Rand, DTL': Random sampling, separation using Decision Tree Learning, 50 samples per refinement iteration, minimizing number of state variables.

4. 'Eqv, DTL': Sampling with equivalence queries, separation using Decision Tree Learning, minimizing number of state variables.

5. 'Eqv, DTL, Inp': Sampling with equivalence queries, separation using Decision Tree Learning, minimizing number of inputs.

For the results in Table 4.2, we added the results of [Chauhan *et al.*, 2002] (they did not report their results for the circuits in the first table).

For each run, we measured the total running time in seconds ('Time'), the number of refinement steps ('S'), and the number of state variables in the final abstract model ('L'). The original number of state variables in each circuit in indicated in its name. A '$-$' symbol indicates run-time longer than 10000 seconds.

The experiments indicate that our technique expedites standard model checking in terms of execution time. We also compared our approach with the technique presented in [Chauhan *et al.*, 2002] (Depen). In most cases our procedure, as expected, constructs a smaller abstract model compared to them (as indicated by the number of state variables chosen). For this set of examples, out approach did not translate in general to faster run times, because the model checking phase for the abstract models was not a bottleneck.

Comparing the various configurations of our tool, it is apparent that in most cases the reduction in the number of required state variables translates to a reduction in the total execution time. There were cases (see, for example, circuit $IU50$ in Table 4.2), however, in which smaller sets of separating variables resulted in longer execution time. Such 'noise' in the experimental results is typical to OBDD-based techniques.

We also tried another set of examples, as summarized in Table 4.3[1]. For this set of examples we replaced Cadence SMV with the model checker used by [Chauhan

[1]Unfortunately we could not compare many other circuits because the model checker used by [Chauhan *et al.*, 2002] is unstable.

Circuit	SMV	Rand, ILP			Rand, DTL			Eqv, DTL			Depen			Eqv, DTL, Inp		
	Time	Time	S	L	Time	S	L	Time	S	L	time	S	L	time	S	L
$IU30$	7.3	8.0	3	20	7.5	3	20	6.5	3	20	**1.9**	4	20	5.56	3	20
$IU35$	19.1	11.8	4	21	12.7	4	21	11.0	4	21	**10.4**	5	21	22.58	4	21
$IU40$	53.6	25.9	6	23	19.0	5	22	16.1	5	22	**13.3**	6	22	33.78	5	22
$IU45$	226.1	28.3	5	22	25.3	5	22	**22.1**	5	22	25	6	22	38.9	5	22
$IU50$	1754	160.4	13	32	85.1	10	27	15120	7	31	**32.8**	6	22	57.39	5	22
$IU55$	-	-	-	-	-	-	-	-	-	-	61.9	4	20	**58.94**	3	20
$IU60$	-	-	-	-	-	-	-	-	-	-	**65.5**	4	20	76.74	3	20
$IU65$	-	-	-	-	-	-	-	-	-	-	**67.5**	4	20	79.99	3	20
$IU70$	-	-	-	-	-	-	-	-	-	-	71.4	4	20	**69.39**	3	20
$IU75$	-	1080	21	38	586.7	16	33	130.5	5	26	**15.7**	5	21	22.59	4	21
$IU80$	-	1136	21	38	552.5	16	33	153.4	5	26	**21.1**	5	21	25.61	4	21
$IU85$	-	1162	21	38	581.2	16	33	167.7	5	26	**24.6**	5	21	27.5	4	21
$IU90$	-	965	20	37	583.3	16	33	167.1	5	26	**24.3**	5	21	27.96	4	21

Table 4.2: Model checking results for property 2.

et al., 2002] to check the abstract models. This model checker is built on top of NuSMV2 and has several optimizations that Cadence SMV doesn't have, like a very efficient mechanism for early quantification, as describes in [Chauhan *et al.*, 2001a,b] (for the smaller examples described in the first two tables changing the model checker did not make any notable difference).

Here we can see that our method performs better in four cases and worse in two. As explained in Section 4.12, we attribute the smaller number of state variables that they find in the last two cases to the arbitrariness of the counterexamples that are generated by the model checker.

		Depen			Eqv, DTL, Inp		
Design	Length	Time	S	L	Time	S	L
M9	TRUE	10.2	2	38	**2.9**	1	38
M6	TRUE	44.3	4	50	**18.8**	4	50
M16	TRUE	1162	61	35	**44.7**	3	34
M17	TRUE	-			**733**	8	39
D6	20	**917**	46	89	1773	43	92
IUp1	TRUE	**3350**	13	19	-	9	41

Table 4.3: Results for various large hardware designs, comparing our techniques with [Chauhan *et al.*, 2002].

Chapter 5

Learning Abstractions without Refinement

5.1 Abstraction as Inductive Learning

As described in Section 3.2, the aim of abstraction is to identify an abstract model that on the one hand is small enough to be handled by the model checker, and on the other hand avoids grouping of concrete states that introduces spurious counterexamples.

Abstraction can be formulated as an inductive learning problem as follows. Abstraction techniques try to infer an abstraction function $h : S \to \hat{S}$ such that the corresponding abstract model does not have any spurious counterexamples. The set of candidate functions over which abstraction searches is implicitly defined by the technique being used to generate the abstract model (Section 3.2). This learner has both forms of inductive biases (Section 2.3).

1. *Restriction Bias:* Abstraction techniques search over a very small subset of the set of all possible abstraction functions (Section 3.3). For example, localization abstraction on a circuit with n boolean latches considers only 2^n abstraction functions (corresponding to the 2^n subsets of the set of n latches). The number of possible abstraction functions for this circuit is much greater than 2^{2^n}.

2. *Preference Bias:* An abstraction function corresponding to an abstract model with fewer states is preferred, because such a model is typically easier to model check.

In order to complete the formulation of abstraction as an inductive learner, we need some notion of *samples* that constitute the experience available to the learner. For this, we introduce *broken traces* on concrete models. Broken traces capture the necessary and sufficient conditions for the existence of an abstract counterexample. The sample set for the learner is a set of broken traces. The abstraction function generated by the learner *eliminates* all the broken traces in the concrete model (Figure 5.1).

5.2 Broken Traces

We define the notion of broken traces for both safety and liveness properties.

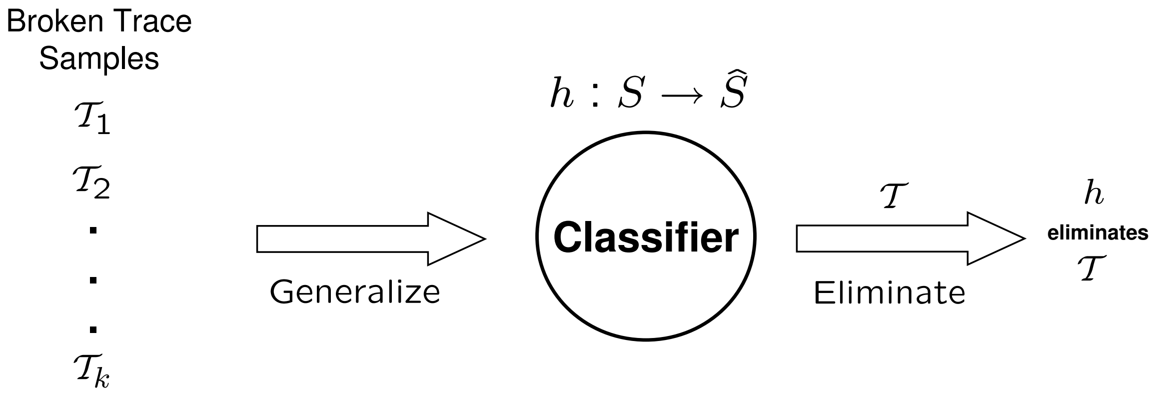

Figure 5.1: Abstraction as Inductive Learning (see Figure 2.1).

5.2.1 Broken Traces for Safety Properties

Definition 5.2.1. Given a model $M = (S, I, R, E, F)$ and an abstraction function h, a *broken trace for safety* $\mathcal{T}$ on M for h is a sequence of pairs of concrete states $\langle (s_1, t_1), \ldots (s_m, t_m) \rangle$, $m \geq 1$ such that

1. $I(s_1)$, i.e. s_1 is an initial state.
2. $\forall 1 \leq i \leq m.\ h(s_i) = h(t_i)$, i.e, s_i and t_i are mapped to the same abstract state by the abstraction function.
3. $\forall 1 \leq i < m.\ R(t_i, s_{i+1})$, i.e, $t_i \rightarrow s_{i+1}$ is a concrete transition.
4. $E(t_m)$, i.e. t_m is an error state.

A broken trace $\langle (s_1, t_1), \ldots (s_m, t_m) \rangle$ is said to *break* at cycle i if $s_i \neq t_i$. If a broken trace has no breaks, it corresponds to a counterexample $C = \langle s_1, \ldots s_m \rangle$ on the

```
MODULE main
VAR
        x, y, z : boolean;
ASSIGN
        init(x) := 0;
        next(x) := x;

        init(y) :=  0;
        next(y) := !y;

        init(z) := 1;
        next(z) := !x | !y;
SPEC AGz
```

Figure 5.2: An example model $\mathcal{N}$ in NuSMV input format.

concrete model. Consider the model $\mathcal{N}$ (shown in Figure 5.2) and the localization abstraction function $h : \{0,1\}^3 \rightarrow \{0,1\}^2$ for $\mathcal{N}$ defined as:

$$h((x, y, z)) = (y, z) \tag{5.1}$$

Figure 5.3 shows a broken trace on $\mathcal{N}$ for h. This broken trace breaks at cycle 2.

5.2.2 Broken Traces for Liveness Properties

Definition 5.2.2. Given a model $M = (S, I, R, E, F)$ and an abstraction function h, a *broken trace for liveness* $\mathcal{T}$ on M for h is a sequence of triplets of concrete states $\langle(s_1, u_1, t_1), \dots (s_l, u_l, t_l), \dots (s_m, u_m, t_m)\rangle$, $m \geq 2$, $1 \leq l < m$, such that

1. $I(s_1)$, i.e. s_1 is an initial state.

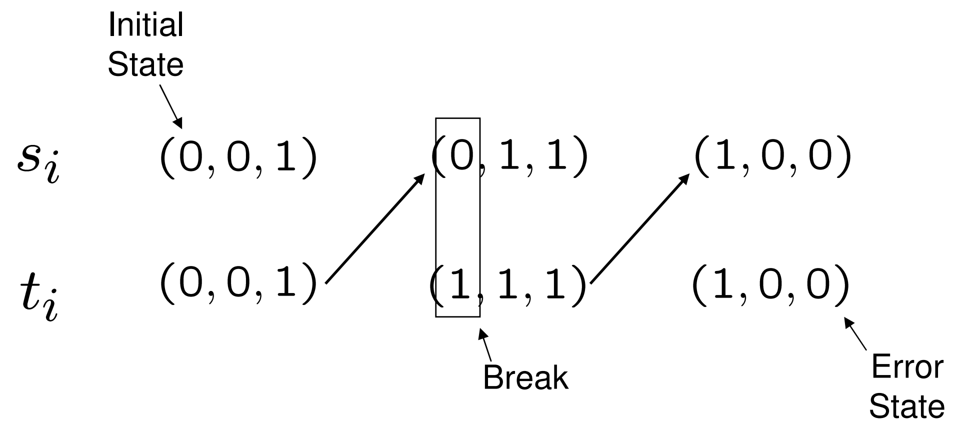

Figure 5.3: A broken trace on $\mathcal{N}$ (Figure 5.2) for the abstraction function defined in Equation 5.1. A state is an assignment to (x, y, z).

2. $\forall 1 \leq i \leq m.\ h(s_i) = h(u_i) = h(t_i)$, i.e, s_i, t_i and u_i are mapped to the same abstract state by the abstraction function.

3. $\forall 1 \leq i < m.\ R(t_i, s_{i+1})$, i.e, $t_i \rightarrow s_{i+1}$ is a concrete transition.

4. $\forall 1 \leq i \leq m.\ \neg F(u_i)$, i.e, u_i is a non-good state.

5. $\exists 1 \leq l < m.\ t_m = s_l$.

5.3 Broken Traces and Abstract Counterexamples

Theorem 5.3.1. *Given a model $M = (S, I, R, E, F)$ and an abstraction function h, there exists a counterexample for the safety property on the abstract model corresponding to h if and only if there exists a broken trace for safety on M for h (See Figure 5.4).*

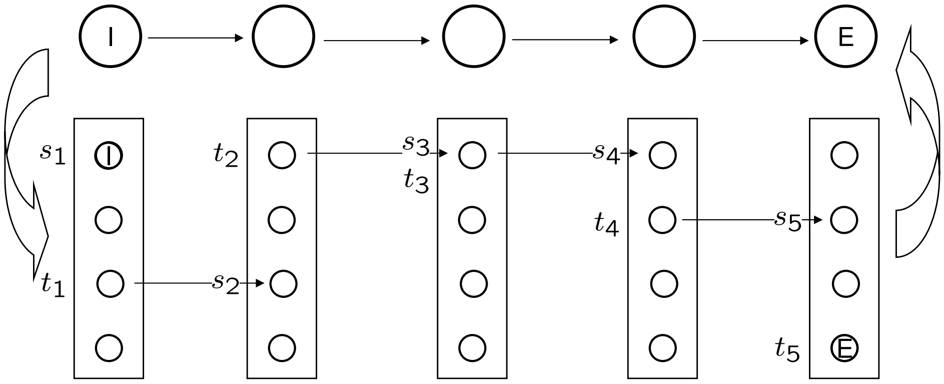

Figure 5.4: An abstract counterexample for a safety property and a broken trace over the corresponding concrete states. This figure illustrates Theorem 5.3.1.

Proof. (IF) Assume that $\mathcal{T}$ is a broken trace for safety on M for h. Let $\mathcal{T} = \langle (s_1, t_1), \ldots (s_m, t_m) \rangle$. Let $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{E}, \hat{F})$ be the abstract model corresponding to h. Let $\hat{C} = \langle \hat{s}_1, \hat{s}_2, \ldots \hat{s}_m \rangle$, where $\hat{s}_i = h(s_i)$. Since $I(s_1)$, by Definition 3.2.2, we have $\hat{I}(\hat{s}_1)$. By Definition 5.2.1, $R(t_i, s_{i+1})$. Therefore, by Definition 3.2.2, $\hat{R}(h(t_i), h(s_{i+1}))$, i.e. $\hat{R}(\hat{s}_i, \hat{s}_{i+1})$. Since $E(t_m)$, by Definition 3.2.2, we have $\hat{E}(\hat{s}_m)$. Hence, $\hat{C}$ is a path counterexample on $\hat{M}$.

(ONLY IF) Assume that $\hat{C}$ is a path counterexample on the abstract model $\hat{M}$ corresponding to h. Let $\hat{C} = \langle \hat{s}_1, \hat{s}_2, \ldots \hat{s}_m \rangle$. Let $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{E}, \hat{F})$. Since $\hat{I}(\hat{s}_1)$, by Definition 3.2.2, there exists a state s_1 such that $I(s_1)$. Since $\hat{R}(\hat{s}_i, \hat{s}_{i+1})$, by Definition 3.2.2, there exist states t_i and s_{i+1} such that $h(t_i) = \hat{s}_i$, $h(s_{i+1}) = \hat{s}_{i+1}$ and $R(t_i, s_{i+1})$. Since $\hat{E}(\hat{s}_m)$, by Definition 3.2.2, there exists t_m such that $E(t_m)$. Thus, $\mathcal{T} = \langle (s_1, t_1), \ldots (s_m, t_m) \rangle$ is a broken trace for safety on M for h. $\square$

For example, consider the abstract model in Figure 5.5 for $\mathcal{N}$, corresponding to the

```
MODULE main
VAR
       y, z : boolean;
IVAR
       x : boolean;
ASSIGN
       init(y) :=  0;
       next(y) := !y;
       init(z) := 1;
       next(z) := !x | !y;
SPEC AGz
```

Figure 5.5: Abstract model for $\mathcal{N}$ (Figure 5.2) corresponding to h defined by Equation 5.1.

abstraction function defined in Equation 5.1. Figure 5.6 shows a counterexample on this model. This abstract counterexample corresponds to the broken trace shown in Figure 5.3.

Theorem 5.3.2. *Given a model $M = (S, I, R, E, F)$ and an abstraction function h, there exists a counterexample for the liveness property on the abstract model corresponding to h if and only if there exists a broken trace for liveness on M for h.*

State: (y,z)

$$(0,1) \longrightarrow (1,1) \longrightarrow \underset{\neg z}{(0,0)}$$

Figure 5.6: A counterexample on the model in Figure 5.5.

Proof. (IF) Assume that $\mathcal{T}$ is a broken trace for liveness on M for h. Let $\mathcal{T} = \langle (s_1, u_1, t_1), \ldots (s_l, u_l, t_l), \ldots (s_m, u_m, t_m) \rangle$. Let $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{E}, \hat{F})$ be the abstract model corresponding to h. Let $\hat{C} = \langle \hat{s}_1, \ldots \hat{s}_l, \ldots \hat{s}_m \rangle$, where $\hat{s}_i = h(s_i)$. Since $I(s_1)$, by Definition 3.2.2, we have $\hat{I}(\hat{s}_1)$. By Definition 5.2.2, $R(t_i, s_{i+1})$. Therefore, by Definition 3.2.2, $\hat{R}(h(t_i), h(s_{i+1}))$, i.e. $\hat{R}(\hat{s}_i, \hat{s}_{i+1})$. Since $\neg F(u_i)$, by Definition 3.2.2, we have $\neg \hat{F}(\hat{s}_i)$. Since $t_m = s_l$, we have $\hat{s}_m = \hat{s}_l$. Hence, $\hat{C} = \langle \hat{s}_1, \ldots (\hat{s}_l, \ldots \hat{s}_{m-1})^\omega \rangle$ is a loop counterexample on $\hat{M}$.

(ONLY IF) Assume that $\hat{C}$ is loop counterexample on the abstract model $\hat{M}$ corresponding to h. Let $\hat{C} = \langle \hat{s}_1, \ldots (\hat{s}_l, \ldots \hat{s}_{m-1})^\omega \rangle$. Let $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{E}, \hat{F})$. Since $\hat{I}(\hat{s}_1)$, by Definition 3.2.2, there exists a state s_1 such that $I(s_1)$. Since $\hat{R}(\hat{s}_i, \hat{s}_{i+1})$, by Definition 3.2.2, there exist states t_i and s_{i+1} such that $h(t_i) = \hat{s}_i$, $h(s_{i+1}) = \hat{s}_{i+1}$ and $R(t_i, s_{i+1})$. Since $\neg \hat{F}(\hat{s}_i)$, by Definition 3.2.2, there exists u_i such that $\neg F(u_i)$. Since $\hat{R}(\hat{s}_{m-1}, \hat{s}_l)$, by Definition 3.2.2, there exist states t_{m-1} and s_m such that $h(t_{m-1}) = \hat{s}_{m-1}$, $h(s_m) = \hat{s}_l$ and $R(t_{m-1}, s_m)$. Thus, $\mathcal{T} = \langle (s_1, u_1, t_1), \ldots (s_l, u_l, t_l), \ldots (s_{m-1}, u_{m-1}, t_{m-1}), (s_m, u_l, t_l) \rangle$ is a broken trace for liveness on M for h. □

5.4 Eliminating Broken Traces

Definition 5.4.1. An abstraction function g, and the corresponding abstract model, are said to *eliminate* a broken trace for safety $\langle (s_1, t_1), \ldots (s_m, t_m) \rangle$ if $\exists 1 \leq i \leq m.$ $g(s_i) \neq g(t_i)$.

For example, the abstraction function $g((x,y,z)) = (x)$ eliminates the broken trace in Figure 5.3, because $g((0,1,1)) = (0)$ and $g((1,1,1)) = (1)$.

Definition 5.4.2. An abstraction function g, and the corresponding abstract model, are said to *eliminate* a broken trace for liveness $\langle(s_1, u_l, t_1), \ldots (s_l, u_l, t_l), \ldots (s_m, u_m, t_m)\rangle$ if $\exists 1 \leq i \leq m.(g(s_i) \neq g(t_i)) \vee (g(s_i) \neq g(u_i)) \vee (g(t_i) \neq g(u_i))$.

5.5 Our Abstraction Strategy

Theorem 5.3.1 and Theorem 5.3.2 say that the existence of a broken trace on a concrete model for an abstraction function h is a *necessary and sufficient* condition for the existence of a counterexample on the abstract model corresponding to h. This is the motivation behind our abstraction strategy. *We compute the smallest abstract model that eliminates all broken traces on the concrete model.* Theorem 5.3.1 and Theorem 5.3.2 imply that this is the smallest abstract model that can prove the property.

5.6 Learning Abstractions

The naive method of computing the abstract model by generating and eliminating all broken traces is infeasible, because the set of broken traces is infinite. Instead, we *learn* this model by generating a set of sample broken traces such that the abstract model that eliminates the broken traces in this set also eliminates all other broken traces.

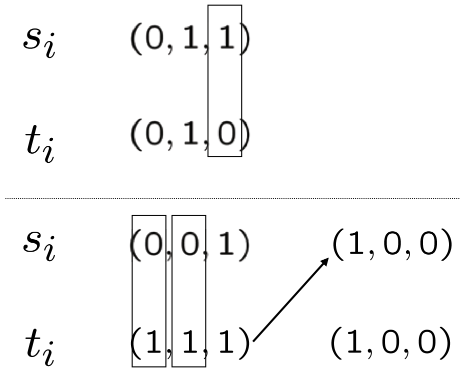

Figure 5.7: Sample broken traces on $\mathcal{N}$ (Figure 5.2). A state is an assignment to (x, y, z).

For example, the broken trace samples in Figure 5.7 are eliminated by the abstraction function

$$h((x, y, z)) = (x, z) \tag{5.2}$$

This abstraction function also eliminates the broken trace in Figure 5.3. The property holds on the corresponding abstract model shown in Figure 5.8, and therefore by Theorem 5.3.1, h eliminates all broken traces in $\mathcal{N}$.

Figure 5.9 is a simplified view of our overall strategy. Starting with an empty set, we iteratively generate the set of broken trace samples. In each iteration of the loop, we compute an abstract model that eliminates all broken traces in the sample set, and then use the counterexample produced by model checking this abstract model to guide the search for new broken traces that are not eliminated by the current abstract model. The call to the model checker is like an equivalence query (Section 2.2) in the inductive learning framework. If the property holds on the abstract model, it implies that our learner has derived the required abstraction function. Otherwise, the broken

```
MODULE main
VAR
    x, z : boolean;
IVAR
    y    : boolean;
ASSIGN
    init(x) := 0;
    next(x) := x;

    init(z) := 1;
    next(z) := !x | !y;
SPEC AGz
```

Figure 5.8: Abstract model for $\mathcal{N}$ (Figure 5.2) corresponding to h defined by Equation 5.2. The property holds on this model.

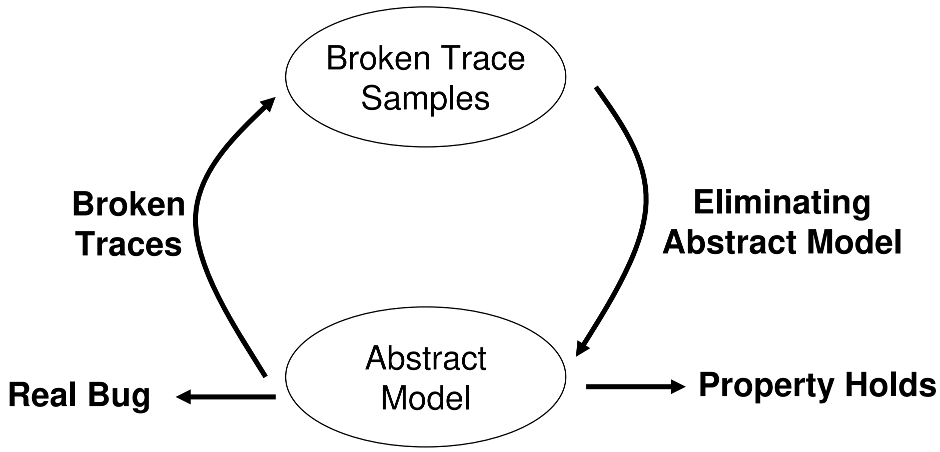

Figure 5.9: Learning Abstract Models.

traces generated are like the objects that are incorrectly classified by the abstraction function generated by the learner. *The abstract model generated in the next iteration from the augmented set of samples is not necessarily a refinement of the previous abstract model.* The loop terminates when either of the following happens:

1. No counterexample is present in the abstract model (property holds).

2. A broken trace with no breaks is generated (property does not hold).

5.6.1 The LearnAbs Algorithm

Algorithm 5.1 Learning Abstractions for Model Checking

LearnAbs (M, φ)

1: $B = \{\}$

2: **while** (1) **do**

3: $\quad h = \text{ComputeAbstractionFunction}(B)$

4: $\quad \hat{M} = \text{BuildAbstractModel}(M, h)$

5: $\quad$ **if** $MC(\hat{M}, \varphi) = TRUE$ **then return** 'TRUE'

6: $\quad$ **else** Let C be the counterexample produced by MC

7: $\quad$ **for** $(n = 1; n \leq N; n = n + 1)$ **do**

8: $\quad\quad \mathcal{T} = \text{GenerateBrokenTrace}(M, C)$

9: $\quad\quad$ **if** $\mathcal{T}$ has no breaks **then return** 'FALSE'

10: $\quad\quad$ **else** $B = B \cup \{\mathcal{T}\}$

The pseudo-code for *Learning Abstractions*(LearnAbs) is shown in Algorithm 5.1. The *ComputeAbstractionFunction* function (line 3) computes an abstraction function that eliminates all broken traces in the sample set B (see Section 5.9). The *Build-AbstractModel* function (line 4) builds the abstract model corresponding to h (see Section 5.7). If the model checker proves the property on the abstract model (line 5), the algorithm returns TRUE. Otherwise, the *GenerateBrokenTrace* function (line 8) generates N broken trace samples corresponding to the abstract counterexample C (see Section 5.10). If a real bug is found in the process (line 9), the algorithm returns FALSE. Otherwise, the loop is repeated with the augmented set of samples (line 10).

5.6.2 Termination

Since the broken trace(s) generated in a particular iteration of the loop are not eliminated by the current abstraction function h, the same abstraction function will not be computed in subsequent iterations. Thus, termination of the LearnAbs is guaranteed if the following conditions are met:

1. The *ComputeAbstractionFunction* function generates abstraction functions from a finite set H.

2. The set H contains the identity abstraction function $h(s) = s$, for which the abstract model is the same as the concrete model.

These assumptions apply to many practical scenarios, including localization abstraction, and predicate abstraction over a finite set of predicates (Section 3.3).

We now describe an implementation of our LearnAbs algorithm that uses localization abstraction for verification of hardware circuits.

5.7 Abstraction Functions

We use localization abstraction based on visible/invisible variables (Section 3.3.2). The *BuildAbstractModel* function simply removes the logic that defines the variables in $\mathcal{I}$, and replaces these variables with inputs (Section 3.3.3). Our initial abstract model corresponds to $\mathcal{V} = \{\}$, i.e. all variables are invisible.

5.8 Model Checker

We can use any off-the-shelf checker for this step. Our implementation interfaces with a state-of-the-art model checker, Cadence SMV [McMillan].

5.9 Computing the Eliminating Abstract Model

For a broken trace for safety $\mathcal{T} = \langle (s_1, t_1), \ldots (s_m, t_m) \rangle$, the *eliminating set* $E_{\mathcal{T}}$ consists of all variables r such that for some $1 \leq i \leq m$, the states s_i and t_i differ on the value of r. Similarly, for a broken trace for liveness $\mathcal{T} = \langle (s_1, u_1, t_1), \ldots (s_m, u_m, t_m) \rangle$, the *eliminating set* $E_{\mathcal{T}}$ consists of all variables r such that for some $1 \leq i \leq m$, the states s_i and t_i, or the states s_i and u_i, or the states u_i and t_i differ on the value of r. The broken trace $\mathcal{T}$ is eliminated by an abstraction function $h^{\mathcal{V}}$ if $E_{\mathcal{T}} \cap \mathcal{V} \neq \{\}$.

For example, the broken traces in Figure 5.7 are eliminated by abstraction functions corresponding to $\mathcal{V} = \{x, z\}$, $\mathcal{V} = \{y, z\}$, and $\mathcal{V} = \{x, y, z\}$. Given a set B of broken trace samples, we want to compute the smallest set $\mathcal{V}$ such that $h^{\mathcal{V}}$ eliminates all broken traces in B. This computation corresponds to the *minimum hitting-set problem*, which is an NP-complete problem. We formulate this as an Integer Linear Program (ILP), and solve it using an ILP-solver. This formulation is very similar to the one described in Section 4.8.1. We also have a formulation that minimizes the number of inputs (instead of the number of state variables) in the abstract model (Section 4.9).

The ILP-based approach computes the smallest eliminating set. However, the complexity of this approach is exponential in the number of state variables. We also implemented various polynomial-time algorithms that compute an approximation of the minimum hitting-set. These algorithms rank variables based on some heuristic measure, and then compute a hitting-set by greedily selecting variables till all the traces are eliminated. For example, one of the heuristic measures we used was the number of breaks on a variable over all the broken trace samples. The greedy algorithms do not guarantee the smallest abstract model, but their complexity is polynomial in the number of state variables and the number of broken trace samples.

5.10 Generating Broken Traces

A SAT-solver implements a function $\textsc{Sat}[F]$ that returns an arbitrary satisfying assignment for the boolean formula F. We enhanced the SAT-solver to implement

'SAT with hints'. The function SATHINT[F,$\mathcal{H}$] takes as input a boolean formula F and a set $\mathcal{H}$ of assignments to a subset of variables in F. It returns a satisfying assignment for F that agrees with the assignments in $\mathcal{H}$ on 'many' variables. This is achieved by forcing the SAT-solver to first decide on the literals corresponding to the variable assignments in $\mathcal{H}$. Thus, the satisfying assignment will disagree with $\mathcal{H}$ on a variable v only if v is forced to a different value by a conflict.

Theorem 5.3.1 and Theorem 5.3.2 say that if there is a counterexample on the abstract model, there exists a broken trace on the concrete model for the corresponding abstraction function. We illustrate how we generate this broken trace for safety properties. The broken traces for liveness are generated in a similar fashion. The *GenerateBrokenTraceS* function (Algorithm 5.2) generates broken traces for safety. Starting with a concrete initial state s_1 (line 2), it successively finds a concrete transition corresponding to each of the abstract transitions in the counterexample (line 4). The hints to SATHINT ensure that at cycle i, a state t_i different from s_i is picked only if s_i does not have a transition to some concrete state in $\hat{s}_{i+1}$. This helps in reducing the size of the eliminating set for the broken trace. A broken trace with a smaller eliminating set is better because it helps the sampling loop to converge faster (Section 4.10). Multiple samples are generated by randomizing the selection of assignments to the inputs.

Note that our approach does not perform BMC on the concrete model. The SAT-solver works on a single frame of the transition relation, thus it can potentially handle much larger designs. We cannot replace the SAT-solver with a circuit simulator, because the circuit outputs (latches) are constrained to lie in the corresponding abstract state in the counterexample. A circuit-simulator, on the other hand, only permits constraints on the inputs. A potential disadvantage of our approach for designs with bugs is that our method for generating broken traces might generate a large number of samples before it finds an unbroken trace corresponding to a bug. In order to deal with this, if the number of samples at a particular depth reaches a threshold, we check for the presence of a real bug at that depth by performing BMC on the concrete model.

Algorithm 5.2 Generating Broken Traces

GenerateBrokenTraceS(M, C)

1: **Let** $C = \langle \hat{s}_1, \ldots \hat{s}_m \rangle$

2: $s_1 = \text{SAT}[\ I(s_1) \wedge (h(s_1) = \hat{s}_1)\ ]$

3: **for** $(i = 1;\ i < m;\ i = i + 1)$ **do**

4: $\quad (t_i, s_{i+1}) = \text{SATHINT}[\ R(t_i, s_{i+1}) \wedge (h(t_i) = \hat{s}_i) \wedge (h(s_{i+1}) = \hat{s}_{i+1}),\ \ \{t_i = s_i\}\ ]$

5: $t_m = \text{SATHINT}[\ E(t_m) \wedge (h(t_m) = \hat{s}_m),\ \ \{t_m = s_m\}\ ]$

6: **Return** $\mathcal{T} = \langle (s_1, t_1), \ldots (s_m, t_m) \rangle$

5.11 LearnAbs vs. Separating Deadend/Bad

Many abstraction techniques eliminate a spurious abstract counterexample as follows: they identify an abstract failure state by a forward (or backward) simulation of the counterexample on the concrete model and then remove the abstract transition from (or to) the failure state by splitting the failure state into multiple abstract states. The drawback of these techniques is that they focus on a single abstract state.

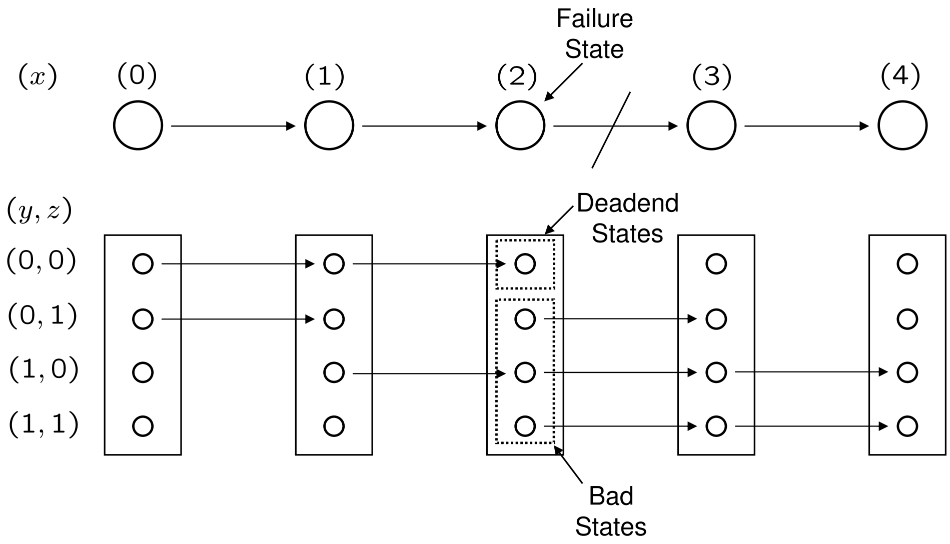

Figure 5.10: $\mathcal{V} = \{x\}$, $\mathcal{I} = \{y, z\}$. The counterexample breaks at abstract state (2).

A smaller abstract model that eliminates the counterexample can be generated by splitting multiple abstract states of the counterexample. Moreover, these techniques do not guarantee that the counterexample is eliminated. We illustrate these shortcomings through some examples in the context of the approach based on *separating deadend and bad states* (Section 4.5). We also illustrate how our approach fixes these shortcomings.

Example 5.11.1. Consider the abstract counterexample in Figure 5.10, with $\mathcal{V} = \{x\}$ and $\mathcal{I} = \{y, z\}$. Figure 5.11 is an abstract model obtained by separating state set S_1 from S_2, and S_3 from S_4, where $S_1 = \{(2, 0, 0)\}$, $S_2 = \{(2, 1, 0), (2, 1, 1)\}$, $S_3 = \{(3, 0, 1)\}$, and $S_4 = \{(3, 1, 0), (3, 1, 1)\}$. These sets are indicated by the dashed boxes in Figure 5.11. S_1 and S_2 lie in the abstract state (2), while S_3 and S_4 lie in the abstract state (3). This separation can be achieved by making y visible, and the new

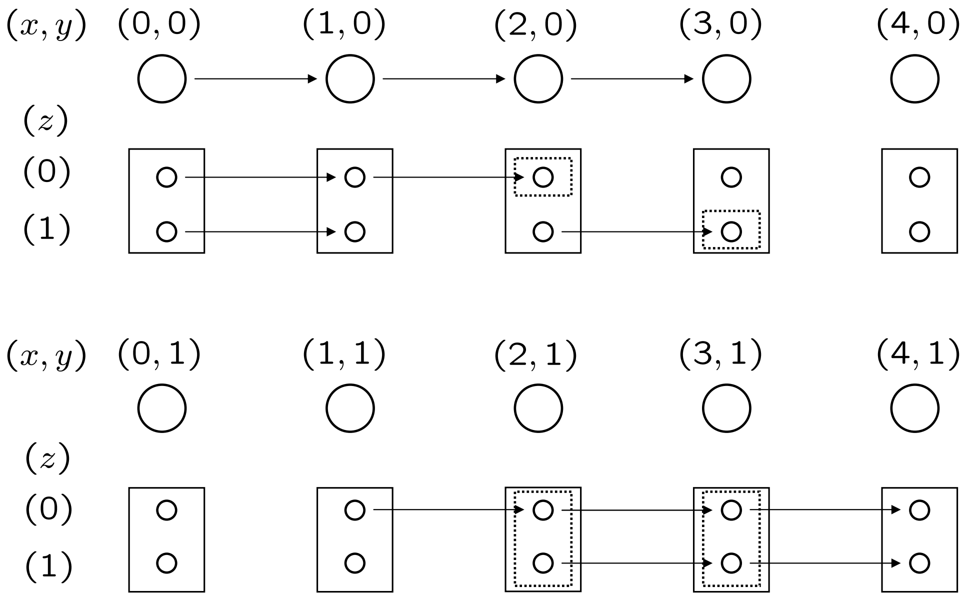

Figure 5.11: An abstract model that eliminates the counterexample in Figure 5.10 by splitting multiple abstract states.

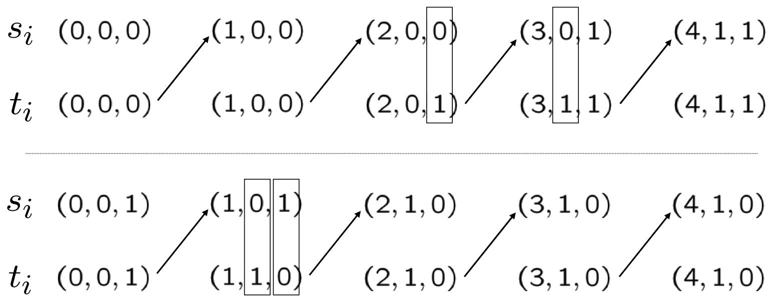

Figure 5.12: Broken trace samples corresponding to the counterexample in Figure 5.10. A state is an assignment to (x, y, z).

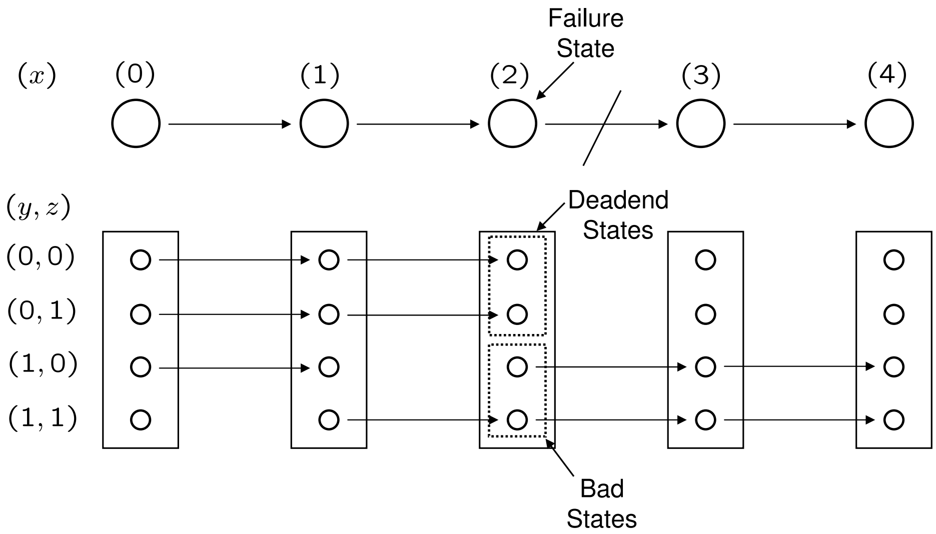

Figure 5.13: $\mathcal{V} = \{x\}$, $\mathcal{I} = \{y, z\}$. The counterexample breaks at abstract state (2).

abstract model eliminates the counterexample. Note that this abstract model does not separate the deadend and bad states. Separating deadend and bad states for this example would require both y and z to be visible, thereby adding unnecessary state variables to the abstract model. This example illustrates that *separating deadend and bad states is not necessary for eliminating the counterexample.* Figure 5.12 illustrates some broken trace samples corresponding to this counterexample. All these samples are eliminated by making y visible. Thus our technique does not have this drawback.

Example 5.11.2. Consider the abstract counterexample in Figure 5.13, with $\mathcal{V} = \{x\}$ and $\mathcal{I} = \{y, z\}$. Figure 5.14 is an abstract model that puts the deadend and bad states into separate abstract states. The counterexample is not eliminated from the new abstract model, because the bad states are reachable in this model. This example

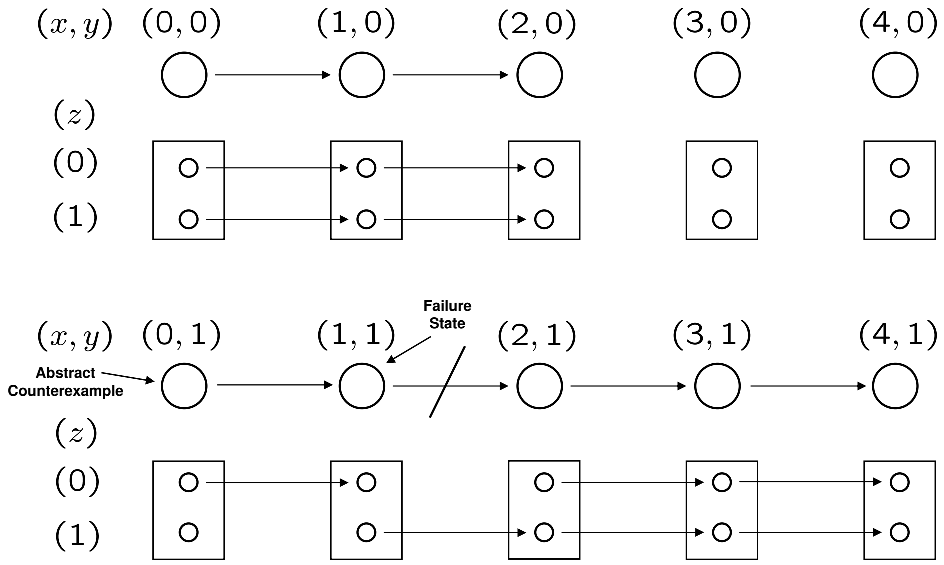

Figure 5.14: An abstract model obtained by separating deadend and bad states in Figure 5.13. The counterexample is not eliminated.

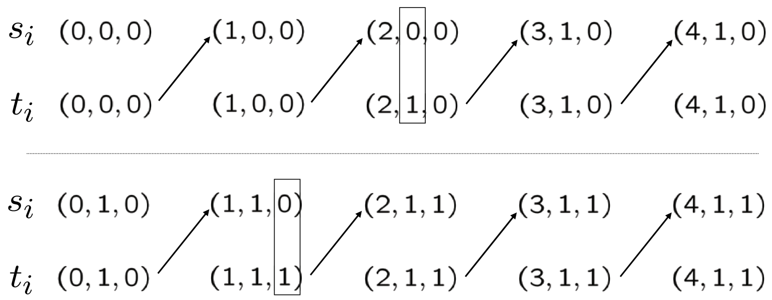

Figure 5.15: Broken trace samples corresponding to the counterexample in Figure 5.13. A state is an assignment to (x, y, z).

illustrates that *separating deadend and bad states is not sufficient to eliminate the counterexample.* Note that the counterexample now fails at an earlier cycle, therefore the counterexample will eventually be eliminated in subsequent refinement iterations. Figure 5.15 illustrates some broken traces corresponding to this counterexample. In order to eliminate these broken traces, we need to make both y and z visible, which eliminates the counterexample. Thus our technique does not suffer from this problem.

5.12 LearnAbs vs. Abstraction Refinement

Counterexample-Guided Abstraction-Refinement (CEGAR) framework (Section 4.1) is a common strategy for automatically generating abstract models. Starting with an initial abstraction function, this technique refines it in each iteration to eliminate one or multiple spurious counterexamples. The smallest abstract model that eliminates a set of counterexamples is not necessarily a refinement of the smallest abstract model that eliminates a subset of this set. Thus, a refinement-based strategy cannot guarantee the smallest abstract model that proves the property. The proof-based abstraction technique presented in [Gupta *et al.*, 2003; McMillan and Amla, 2003] tries to alleviate this problem by building a fresh abstract model at each iteration. They perform BMC on the concrete model up to the length of the counterexample. The abstract model consists of the gates that are used by the SAT-solver to prove unsatisfiability of the BMC instance. However, this approach is also not optimal. The LearnAbs algorithm is not based on refinement, and therefore it does not have this drawback.

Example 5.12.1. Consider the application of a CEGAR-based strategy to $\mathcal{N}$ (Figure 5.2). In the first iteration of the CEGAR loop, an abstract model is generated to eliminate counterexamples of length 1. This could produce an abstract model with $\mathcal{V} = \{y, z\}$ (see Figure 5.5), which is one of the smallest models that eliminates all counterexamples of length 1. In the next iteration, a counterexample of length 2 will be generated and this adds the variable x to $\mathcal{V}$. At this point, the variable y is not needed, but it ends up being part of the abstract model because CEGAR is based on refinement. Even if a fresh abstract model is generated using a SAT-solver, the abstract model will contain the variable y if the SAT-solver first makes decisions on the variables corresponding to y. Our approach, on the other hand, guarantees that y is not included in the final abstract model (see Figure 5.8).

5.13 Example

In this section, we will use the model $\mathcal{M}$ described in Section 3.4 to illustrate the following:

- The working of our LearnAbs algorithm.
- The advantages of LearnAbs over abstraction-refinement.

In Section 4.11, we went over each step of our CEGAR implementation on the same model. At some of the refinement steps (namely Iteration 2 and Iteration 3), there were multiple options available to CEGAR for which variables to add to the abstract model. All these options corresponded to separating sets of the same (smallest) size, and therefore CEGAR was free to choose any one of them. We presented a run in which CEGAR picked the variables that generated an optimal abstract model in the end. We now describe another possible execution sequence which shows how CEGAR can end up making the wrong choices, thereby generating long spurious abstract counterexamples and a large abstract model.

5.13.1 An execution of CEGAR

The deadend and bad states in the sample sets shown below are assignments to the state variables $\{x, y, z, u, c_0.v, c_1.v, c_2.v\}$. In the sample sets, the assignments to the variables in the separating set are shown in a bold font.

- *Abstract (Iteration* 1*) :* The initial abstract model $\mathcal{M}_1$, corresponding to $\mathcal{V} = \{\}$, is shown in Figure 5.16.
- *Model Check (Iteration* 1*) :* Model checking $\mathcal{M}_1$ produces the counterexample shown in Figure 5.17.
- *Spurious? (Iteration* 1*) :* The formula ϕ_{C_S} (Equation 4.2) for the counterexample in Figure 5.17 is unsatisfiable. Thus, the counterexample in spurious.
- *Refine (Iteration* 1*) :* The counterexample is of length $k = 0$. Hence, the failure index f is 0. The final sample sets and the separating set generated by SAMPLESEP are:

$$S_D = \{(1, 0, 0, \mathbf{1}, 0, 0, 0)\}$$

MODULE `main`

IVAR

$\quad i,\ j,\ x,\ y,\ z,\ u,\ c_0,\ c_1,\ c_2\ :$ **boolean**;

SPEC $\mathbf{AG}u$

Figure 5.16: The initial abstract model $\mathcal{M}_1$.

State: ()

()

$\neg u$

Figure 5.17: Counterexample on model $\mathcal{M}_1$ (Figure 5.16).

$$S_B = \{(1, 0, 0, \mathbf{0}, 0, 0, 0)\}$$

$$SepS = \{u\}$$

- *Abstract (Iteration* 2*) :* The abstract model $\mathcal{M}_2$ for iteration 2, corresponding to $\mathcal{V} = \{u\}$, is shown in Figure 5.18.

- *Model Check (Iteration* 2*) :* Model checking $\mathcal{M}_2$ produces the counterexample shown in Figure 5.19.

- *Spurious*? *(Iteration* 2*) :* The formula ϕ_{C_S} (Equation 4.2) for the counterexample in Figure 5.19 is unsatisfiable. Thus, the counterexample in spurious.

```
MODULE main
VAR
        u : boolean;
IVAR
        i, j, x, y, z, c0, c1, c2 : boolean;
ASSIGN
        init(u) := 1;
        next(u) := case
                      c0 & c1 & c2 : x | y | z;
                      1 : u;
                    esac;
SPEC AGu
```

Figure 5.18: The abstract model $\mathcal{M}_2$.

- *Refine (Iteration* 2*) :* The largest index i for which the formula $\mathcal{D}^i_{\mathcal{S}}$ (Equation 4.8) is satisfiable is 0. Hence, the failure index f is 0. The final sample sets and the separating set generated by SampleSep are:

$$S_D = \{(1,0,0,1,\mathbf{0},0,0),(0,1,0,1,\mathbf{0},0,0)\}$$

$$S_B = \{(0,0,0,1,\mathbf{1},1,1)\}$$

$$SepS = \{c_0.v\}$$

- *Abstract (Iteration* 3*) :* The abstract model $\mathcal{M}_3$ for iteration 3, corresponding to $\mathcal{V} = \{u, c_0.v\}$, is shown in Figure 5.20.

State: (u)

$$\begin{array}{ccc} (1) & \longrightarrow & (0) \\ & & \neg u \end{array}$$

Figure 5.19: Counterexample on model $\mathcal{M}_2$ (Figure 5.18).

- *Model Check (Iteration* 3*) :* Model checking $\mathcal{M}_3$ produces the counterexample shown in Figure 5.21.

- *Spurious*? *(Iteration* 3*) :* The formula $\phi_{C_\mathcal{S}}$ (Equation 4.2) for the counterexample in Figure 5.21 is unsatisfiable. Thus, the counterexample in spurious.

- *Refine (Iteration* 3*) :* The largest index i for which the formula $\mathcal{D}^i_\mathcal{S}$ (Equation 4.8) is satisfiable is 1. Hence, the failure index f is 1. The final sample sets and the separating set generated by SampleSep are:

$$S_D = \{(0, 1, 1, 1, 1, \mathbf{0}, 0)\}$$

$$S_B = \{(0, 0, 0, 1, 1, \mathbf{1}, 1)\}$$

$$SepS = \{c_1.v\}$$

- *Abstract (Iteration* 4*) :* The abstract model $\mathcal{M}_4$ for iteration 4, corresponding to $\mathcal{V} = \{u, c_0.v, c_1.v\}$, is shown in Figure 5.22.

- *Model Check (Iteration* 4*) :* Model checking $\mathcal{M}_4$ produces the counterexample shown in Figure 5.23.

```
MODULE main
VAR
        u : boolean;
        c0 : counter_cell(1);
IVAR
        i, j, x, y, z, c1, c2 : boolean;
ASSIGN
        init(u) := 1;
        next(u) := case
                        c0.v & c1 & c2 : x | y | z;
                        1 : u;
                   esac;
SPEC AG u

MODULE counter_cell(cin)
VAR
        v : boolean;
ASSIGN
       init(v) := 0;
       next(v) := v + cin mod 2;
DEFINE
       cout := v & cin;
```

Figure 5.20: The abstract model $\mathcal{M}_3$.

State: $(u, c_0.v)$

$$(1,0) \longrightarrow (1,1) \longrightarrow \underset{\neg u}{(0,0)}$$

Figure 5.21: Counterexample on model $\mathcal{M}_3$ (Figure 5.20).

- *Spurious? (Iteration* 4*) :* The formula $\phi_{C_{\mathcal{S}}}$ (Equation 4.2) for the counterexample in Figure 5.23 is unsatisfiable. Thus, the counterexample in spurious.

- *Refine (Iteration* 4*) :* The largest index i for which the formula $\mathcal{D}^i_{\mathcal{S}}$ (Equation 4.8) is satisfiable is 3. Hence, the failure index f is 3. The final sample sets and the separating set generated by SampleSep are:

$$S_D = \{(1,0,1,1,1,1,\mathbf{0}),(0,1,1,1,1,1,\mathbf{0})\}$$

$$S_B = \{(0,0,0,1,1,1,\mathbf{1})\}$$

$$SepS = \{c_2.v\}$$

- *Abstract (Iteration* 5*) :* The abstract model $\mathcal{M}_5$ for iteration 5, corresponding to $\mathcal{V} = \{u, c_0.v, c_1.v, c_2.v\}$, is shown in Figure 5.24.

- *Model Check (Iteration* 5*) :* Model checking $\mathcal{M}_5$ produces the counterexample shown in Figure 5.25.

- *Spurious? (Iteration* 5*) :* The formula $\phi_{C_{\mathcal{S}}}$ (Equation 4.2) for the counterexample in Figure 5.25 is unsatisfiable. Thus, the counterexample in spurious.

- *Refine (Iteration* 5*) :* The largest index i for which the formula $\mathcal{D}^i_{\mathcal{S}}$ (Equation 4.8) is satisfiable is 7. Hence, the failure index f is 7. The final sample sets and the separating set generated by SampleSep are:

$$S_D = \{(1,0,\mathbf{1},1,1,1,1),(0,1,\mathbf{1},1,1,1,1)\}$$

```
MODULE main
VAR
      u : boolean;
      c0 : counter_cell(1);
      c1 : counter_cell(c0.cout);
IVAR
      i, j, x, y, z, c2 : boolean;
ASSIGN
      init(u) := 1;
      next(u) := case
                   c0.v & c1.v & c2 : x | y | z;
                   1 : u;
                 esac;
SPEC AG u

MODULE counter_cell(cin)
VAR
      v : boolean;
ASSIGN
     init(v) := 0;
     next(v) := v + cin mod 2;
DEFINE
     cout := v & cin;
```

Figure 5.22: The abstract model $\mathcal{M}_4$.

State: $(u, c_0.v, c_1.v)$

$$(1,0,0) \longrightarrow (1,1,0) \longrightarrow (1,0,1) \longrightarrow (1,1,1) \longrightarrow \underset{\neg u}{(0,0,0)}$$

Figure 5.23: Counterexample on model $\mathcal{M}_4$ (Figure 5.22).

$$S_B = \{(0, 0, \mathbf{0}, 1, 1, 1, 1)\}$$

$$SepS = \{z\}$$

- *Abstract (Iteration* 6*) :* The abstract model $\mathcal{M}_6$ for iteration 6, corresponding to $\mathcal{V} = \{z, u, c_0.v, c_1.v, c_2.v\}$, is shown in Figure 5.26.
- *Model Check (Iteration* 6*) :* The property holds on $\mathcal{M}_6$, and therefore the property holds on $\mathcal{M}$. This terminates the CEGAR loop.

The final abstract model that CEGAR generates contains 5 state variables, while the smallest abstract model that proves the property contains only 3 state variables. Moreover, the longest counterexample generated by CEGAR is of length 8, which means that it builds a BMC unfolding of length 8 to check the validity of this counterexample.

5.13.2 An execution of LearnAbs

We now show how LearnAbs behaves on $\mathcal{M}$. For each iteration, we show the abstract model, the abstract counterexample, and the broken trace samples that

```
MODULE main
VAR
      u : boolean;
      c0 : counter_cell(1);
      c1 : counter_cell(c0.cout);
      c2 : counter_cell(c1.cout);
IVAR
      i, j, x, y, z : boolean;
ASSIGN
      init(u) := 1;
      next(u) := case
                   c0.v & c1.v & c2.v : x | y | z;
                   1 : u;
                 esac;
SPEC AGu
MODULE counter_cell(cin)
VAR
      v : boolean;
ASSIGN
     init(v) := 0;
     next(v) := v + cin mod 2;
DEFINE
     cout := v & cin;
```

Figure 5.24: The abstract model $\mathcal{M}_5$.

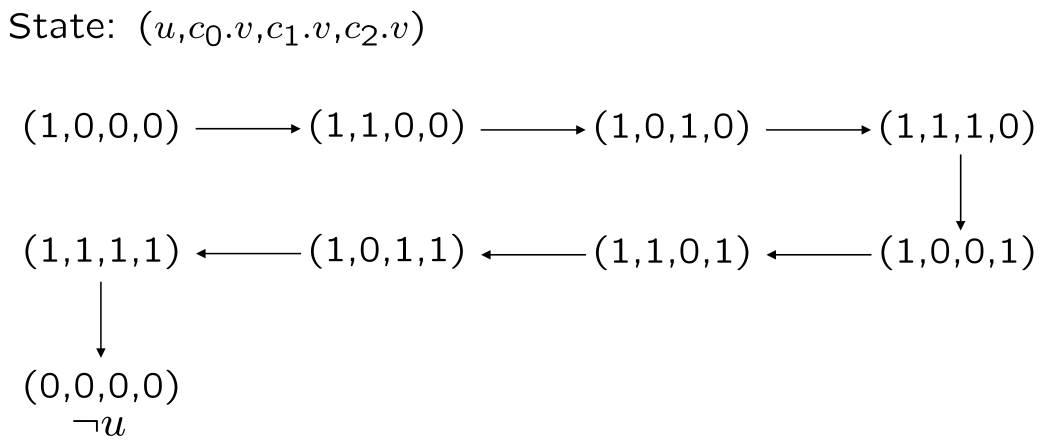

Figure 5.25: Counterexample on model $\mathcal{M}_5$ (Figure 5.24).

LearnAbs generates.

- *Abstract Model (Iteration* 1*) :* Initially, the set of broken trace samples is empty. The initial abstract model $\mathcal{M}_1$, corresponding to $\mathcal{V} = \{\}$, is shown in Figure 5.27.

- *Model Check (Iteration* 1*) :* Model checking $\mathcal{M}_1$ produces the counterexample shown in Figure 5.28.

- *Broken Trace Samples (Iteration* 1*) :* Figure 5.29 shows the broken trace samples generated by LearnAbs. These samples can be eliminated by making u visible.

- *Abstract Model (Iteration* 2*) :* The abstract model $\mathcal{M}_2$ for iteration 2, corresponding to $\mathcal{V} = \{u\}$, is shown in Figure 5.30.

```
MODULE main
VAR
      z, u : boolean;
      c0 : counter_cell(1);
      c1 : counter_cell(c0.cout);
      c2 : counter_cell(c1.cout);
IVAR
      i, j, x, y : boolean;
ASSIGN
      init(z) := 0;
      next(z) := !z;
      init(u) := 1;
      next(u) := case
                    c0.v & c1.v & c2.v : x | y | z;
                    1 : u;
                   esac;
SPEC AGu
MODULE counter_cell(cin)
VAR
      v : boolean;
ASSIGN
     init(v) := 0;
     next(v) := v + cin mod 2;
DEFINE
     cout := v & cin;
```

Figure 5.26: The abstract model $\mathcal{M}_6$. The property holds on this model.

```
MODULE main
IVAR
        i, j, x, y, z, u, c0, c1, c2 : boolean;
SPEC AG u
```

Figure 5.27: The initial abstract model $\mathcal{M}_1$.

State: ()

()
$\neg u$

Figure 5.28: Counterexample on model $\mathcal{M}_1$ (Figure 5.27).

s_i $(0, 1, 0, \boxed{1}, 0, 0, 0)$

t_i $(0, 1, 0, \boxed{0}, 0, 0, 0)$

Figure 5.29: Broken trace samples corresponding to the counterexample in Figure 5.28. Each state is an assignment to $\{x, y, z, u, c_0.v, c_1.v, c_2.v\}$.

```
MODULE main
VAR
        u : boolean;
IVAR
        i, j, x, y, z, c0, c1, c2 : boolean;
ASSIGN
        init(u) := 1;
        next(u) := case
                        c0 & c1 & c2 : x | y | z;
                        1 : u;
                       esac;
SPEC  AGu
```

Figure 5.30: The abstract model $\mathcal{M}_2$.

- *Model Check (Iteration 2) :* Model checking $\mathcal{M}_2$ produces the counterexample shown in Figure 5.31.

- *Broken Trace Samples (Iteration 2) :* Figure 5.32 shows the broken trace samples generated by LearnAbs. All the broken traces seen so far can be eliminated by making u and $c_0.v$ visible.

- *Abstract Model (Iteration 3) :* The abstract model $\mathcal{M}_3$ for iteration 3, corresponding to $\mathcal{V} = \{u, c_0.v\}$, is shown in Figure 5.33.

- *Model Check (Iteration 3) :* Model checking $\mathcal{M}_3$ produces the counterexample shown in Figure 5.34.

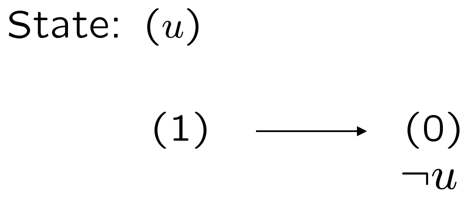

Figure 5.31: Counterexample on model $\mathcal{M}_2$ (Figure 5.30).

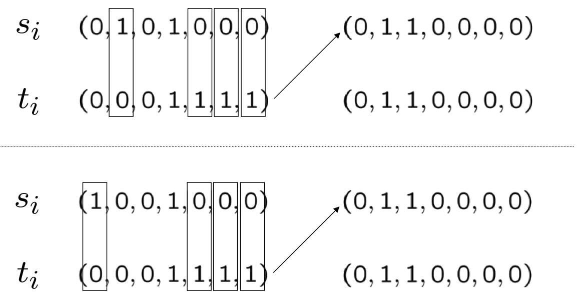

Figure 5.32: Broken trace samples corresponding to the counterexample in Figure 5.31. Each state is an assignment to $\{x, y, z, u, c_0.v, c_1.v, c_2.v\}$.

- *Broken Trace Samples (Iteration* 3*) :* Figure 5.35 shows the broken trace samples generated by LearnAbs. All the broken traces seen so far can be eliminated by making u and $c_1.v$ visible.

- *Abstract Model (Iteration* 4*) :* The abstract model $\mathcal{M}_4$ for iteration 4, corresponding to $\mathcal{V} = \{u, c_1.v\}$, is shown in Figure 5.36.

- *Model Check (Iteration* 4*) :* Model checking $\mathcal{M}_4$ produces the counterexample shown in Figure 5.37.

- *Broken Trace Samples (Iteration* 4*) :* Figure 5.38 shows the broken trace samples generated by LearnAbs. All the broken traces seen so far can be eliminated by making u and $c_2.v$ visible.

- *Abstract Model (Iteration* 5*) :* The abstract model $\mathcal{M}_5$ for iteration 5, corresponding to $\mathcal{V} = \{u, c_2.v\}$, is shown in Figure 5.39.

- *Model Check (Iteration* 5*) :* Model checking $\mathcal{M}_5$ produces the counterexample shown in Figure 5.40.

- *Broken Trace Samples (Iteration* 5*) :* Figure 5.41 shows the broken trace samples generated by LearnAbs. The set of all generated broken trace samples cannot be eliminated by making 2 variables visible. There are 5 3-variable options for LearnAbs:

$$\mathcal{V}_1 = \{u, x, y\}$$

```
MODULE main
VAR
      u : boolean;
      c0 : counter_cell(1);
IVAR
      i, j, x, y, z, c1, c2 : boolean;
ASSIGN
      init(u) := 1;
      next(u) := case
                    c0.v & c1 & c2 : x | y | z;
                    1 : u;
                  esac;
SPEC AG u

MODULE counter_cell(cin)
VAR
      v : boolean;
ASSIGN
      init(v) := 0;
      next(v) := v + cin mod 2;
DEFINE
      cout := v & cin;
```

Figure 5.33: The abstract model $\mathcal{M}_3$.

State: $(u, c_0.v)$

$$(1,0) \longrightarrow (1,1) \longrightarrow \underset{\neg u}{(0,0)}$$

Figure 5.34: Counterexample on model $\mathcal{M}_3$ (Figure 5.33).

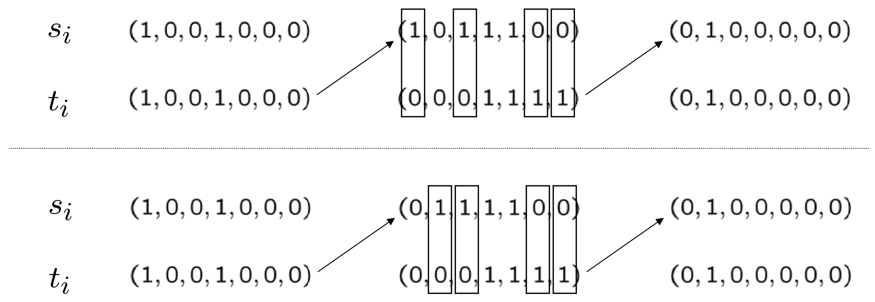

Figure 5.35: Broken trace samples corresponding to the counterexample in Figure 5.34. Each state is an assignment to $\{x, y, z, u, c_0.v, c_1.v, c_2.v\}$.

$$\mathcal{V}_2 = \{u, c_0.v, z\}$$

$$\mathcal{V}_3 = \{u, c_0.v, c_1.v\}$$

$$\mathcal{V}_4 = \{u, c_1.v, c_2.v\}$$

$$\mathcal{V}_5 = \{u, c_2.v, c_3.v\}$$

If LearnAbs picks $\mathcal{V}_1$ or $\mathcal{V}_2$, the resulting abstract model proves the property. In the worst case, LearnAbs ends up picking one of $\mathcal{V}_3$, $\mathcal{V}_4$ or $\mathcal{V}_5$. Lets assume that it picks up $\mathcal{V}_4$.

- *Abstract Model (Iteration* 6*) :* The abstract model $\mathcal{M}_6$ for iteration 6, corresponding to $\mathcal{V} = \{u, c_1.v, c_2.v\}$, is shown in Figure 5.42.
- *Model Check (Iteration* 6*) :* Model checking $\mathcal{M}_6$ produces the counterexample shown in Figure 5.43.
- *Broken Trace Samples (Iteration* 6*) :* Figure 5.44 shows the broken trace samples generated by LearnAbs. The set of samples can be eliminated by $\mathcal{V}_1$,

```
MODULE main
VAR
      u : boolean;
      c1 : counter_cell(p);
IVAR
      i, j, x, y, z, c0, c2, p : boolean;
ASSIGN
      init(u) := 1;
      next(u) := case
                    c0 & c1.v & c2 : x | y | z;
                    1 : u;
                  esac;
SPEC AGu

MODULE counter_cell(cin)
VAR
      v : boolean;
ASSIGN
      init(v) := 0;
      next(v) := v + cin mod 2;
DEFINE
      cout := v & cin;
```

Figure 5.36: The abstract model $\mathcal{M}_4$.

State: $(u, c_1.v)$

$$(1,0) \longrightarrow (1,1) \longrightarrow \underset{\neg u}{(0,0)}$$

Figure 5.37: Counterexample on model $\mathcal{M}_4$ (Figure 5.36).

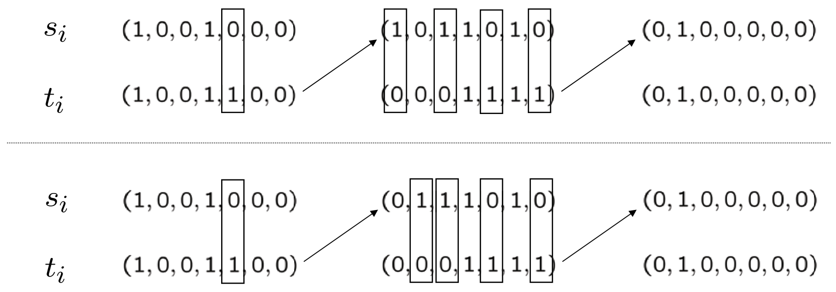

Figure 5.38: Broken trace samples corresponding to the counterexample in Figure 5.37. Each state is an assignment to $\{x, y, z, u, c_0.v, c_1.v, c_2.v\}$.

> $\mathcal{V}_2$, $\mathcal{V}_3$ or $\mathcal{V}_5$. In the worst case, there are two more iterations of the LearnAbs loop, one with $\mathcal{V} = \mathcal{V}_3$ and the other with $\mathcal{V} = \mathcal{V}_4$. Both these iterations produce a counterexample of length 4. Eventually, LearnAbs picks $\mathcal{V}_1$ or $\mathcal{V}_2$ as the set of visible variables and proves the property. The model $\mathcal{M}_7$ corresponding to $\mathcal{V} = \mathcal{V}_1$ is shown in Figure 5.45.

The final abstract model that LearnAbs generates contains 3 state variables. This is one of the smallest models that can prove the property. In each iteration, LearnAbs generates an abstract model that has no more than 3 state variables. Also, the longest counterexample generated by LearnAbs is of length 4. This example illustrates the advantages that LearnAbs has over cegar.

5.14 Optimizations

The following optimizations were implemented on top of the basic LearnAbs algorithm.

```
MODULE main
VAR
       u : boolean;
       c2 : counter_cell(p);
IVAR
       i, j, x, y, z, c0, c1, p : boolean;
ASSIGN
       init(u) := 1;
       next(u) := case
                     c0 & c1 & c2.v : x | y | z;
                     1 : u;
                   esac;
SPEC AGu

MODULE counter_cell(cin)
VAR
       v : boolean;
ASSIGN
      init(v) := 0;
      next(v) := v + cin mod 2;
DEFINE
      cout := v & cin;
```

Figure 5.39: The abstract model $\mathcal{M}_5$.

State: $(u, c_2.v)$

$$(1,0) \longrightarrow (1,1) \longrightarrow \underset{\neg u}{(0,0)}$$

Figure 5.40: Counterexample on model $\mathcal{M}_5$ (Figure 5.39).

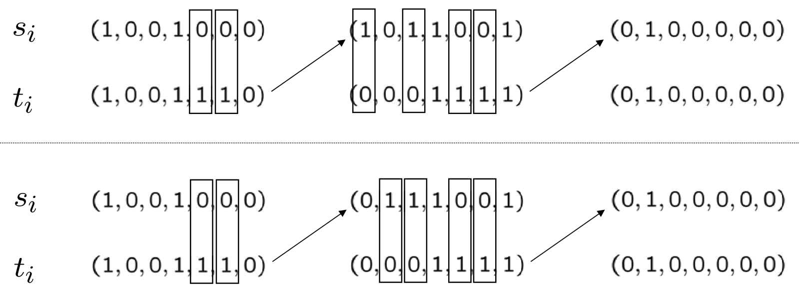

Figure 5.41: Broken trace samples corresponding to the counterexample in Figure 5.40. Each state is an assignment to $\{x, y, z, u, c_0.v, c_1.v, c_2.v\}$.

Fine Grained Abstractions : The latch-level abstraction can be too coarse for larger circuits. We modified the *BuildAbstractModel* function (line 4) to perform BMC on the latch abstraction computed from the samples. The BMC instance is restricted with values from the counterexample. If the BMC instance is unsatisfiable, only the gates used in the proof of unsatisfiability are added to the abstract model [McMillan and Amla, 2003]. If the BMC instance is satisfiable, it means that the counterexample has not been eliminated, and more samples are generated.

Eliminating All Counterexamples : Eliminating one counterexample in each iteration could lead to a lot of expensive model checking calls [Amla and McMillan, 2004]. The LearnAbs loop can be modified to eliminate all counterexamples at the current depth. The modified loop performs (unrestricted) BMC on the current abstract model in the *BuildAbstractModel* function, and proceeds to the model checking step only if the BMC instance is unsatisfiable. If BMC produces a counterexample, the counterexample is used to generate more samples. This optimization assumes

```
MODULE main
VAR
      u : boolean;
      c1 : counter_cell(p);
      c2 : counter_cell(c1.cout);
IVAR
      i, j, x, y, z, c0, p : boolean;
ASSIGN
      init(u) := 1;
      next(u) := case
                    c0 & c1.v & c2.v : x | y | z;
                    1 : u;
                  esac;
SPEC AGu

MODULE counter_cell(cin)
VAR
      v : boolean;
ASSIGN
     init(v) := 0;
     next(v) := v + cin mod 2;
DEFINE
     cout := v & cin;
```

Figure 5.42: The abstract model $\mathcal{M}_6$.

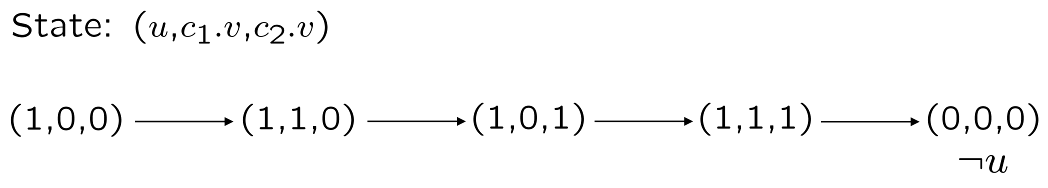

Figure 5.43: Counterexample on model $\mathcal{M}_6$ (Figure 5.42).

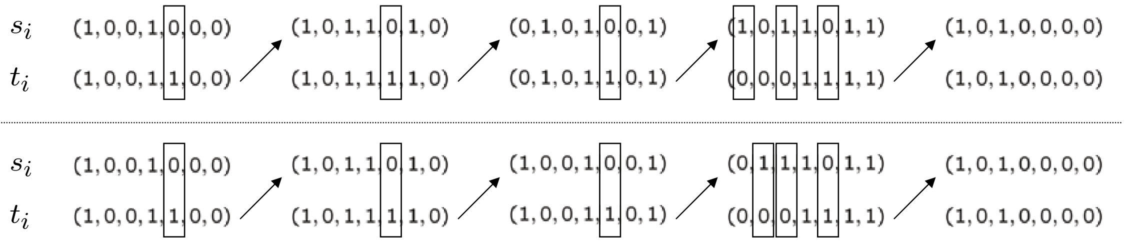

Figure 5.44: Broken trace samples corresponding to the counterexample in Figure 5.43. Each state is an assignment to $\{x, y, z, u, c_0.v, c_1.v, c_2.v\}$.

```
MODULE main
VAR
      x, y, u : boolean;
IVAR
      i, j, z, c0, c1, c2 : boolean;
ASSIGN
      init(x) := j;
      next(x) := i;

      init(y) := !j;
      next(y) := !i;

      init(u) := 1;
      next(u) := case
                     c0 & c1 & c2 : x | y | z;
                     1 : u;
                   esac;
SPEC AGu
```

Figure 5.45: The abstract model $\mathcal{M}_7$. The property hold on this model.

that BMC is faster than model checking.

Reducing Number of Samples : It can be shown that computing the smallest abstract model that eliminates all counterexamples up to a certain length, is a Σ_2^P-complete problem [Umans, 1998] (see Chapter 6 for details). The LEARNABS algorithm could potentially generate a large number of samples before it terminates. In order to balance the time spent in computing the abstract model and in model checking, we added some simple heuristics to the *ComputeAbstractionFunction* function (line 3) to pick larger non-optimal separating sets if the sampling step takes too much time.

Is Small Always Good? : In most cases, an abstract model with fewer state variables is easier to model check. However, for some circuits, we observed that reducing the number of state variables makes model checking harder. The reason for this is as follows: the property can be proved on these circuits using two different sub-circuits, S_1 and S_2. The sub-circuit S_1 contains fewer state variables compared to S_2, however, the combinational logic that S_1 contains is more complex than S_2, and therefore S_1 is harder to model check. Reducing the number of state variables generates the abstract model corresponding to S_1, instead of S_2. After analyzing the high-level descriptions of these circuits, we observed that the properties were derived from compositional verification. At the high-level, these properties were assertions of the form $(A \rightarrow B)$, where the consequent B is true independent of the antecedent A. However, using the antecedent A made the proof of the assertion easier. By forcing our approach to use the state variables in A, we were able to reduce the complexity of model checking. Since the state variables in A were part of the property specification, we achieved this by forcing LearnAbs to always add the property variables to the abstract model, even if the property can be proved without using them.

5.15 Related Work

Starting with Kurshan's *localization reduction* [Kurshan, 1995], there has been a lot of work in automatically generating good abstract models [Barner *et al.*, 2002; Chauhan *et al.*, 2002; Clarke *et al.*, 2003, 2004; Das and Dill, 2002; Glusman *et al.*, 2003; Gupta *et al.*, 2003; Jain *et al.*, 2005; Mang and Ho, 2004; Wang *et al.*, 2001, 2003, 2004a]. Many of these techniques follow the Counterexample-Guided Abstraction-Refinement (cegar) framework (Section 4.1). These techniques start with an initial abstract model and iteratively *add more constraints (refinement)* to eliminate spurious counterexamples, until the property holds on the abstract model or a counterexample is found on the concrete model. The refinement of the abstract model and the search for a concrete counterexample are guided by the abstract counterexamples produced by the model checker.

The technique in [Barner *et al.*, 2002; Chauhan *et al.*, 2002; Clarke *et al.*, 2003, 2004; Gupta *et al.*, 2003] uses BDDs or SAT-solvers to identify the *failure* state, which is the last state in the longest prefix of the abstract counterexample that has a corresponding path in the concrete model. It then adds a set of constraints that eliminate the abstract transition from the failure state by splitting it into multiple abstract states. The methods in [Das and Dill, 2002; Jain *et al.*, 2005] are similar, except that instead of the longest prefix, they look for a minimal spurious sub-trace [Das and Dill, 2002] or the longest suffix [Jain *et al.*, 2005]. The drawback of these strategies is that they focus their efforts on a single abstract state instead of the whole counterexample. A smaller abstract model that eliminates the counterexample can be generated by splitting multiple abstract state in the counterexample. Moreover, identification of the failure state involves building an unrolling of the concrete model, which is expensive. Our approach fixes these drawbacks by analyzing all the abstract states and by never building an unfolding of the concrete model.

The techniques presented in [Mang and Ho, 2004; Wang *et al.*, 2003] use a *game-theoretic* approach to eliminate spurious counterexamples. They analyze the abstract model to identify the variables that can steer the abstract model away from the error states. The approach in [Wang *et al.*, 2001] simulates the abstract counterexample on the concrete model using 3-valued simulation, and looks for variables that conflict with their values in the counterexample. The method in [Glusman *et al.*, 2003] finds variables that are assigned the same value in multiple counterexamples. All these approaches use a heuristic to identify a set of candidate variables to add to the abstract model, and then greedily add variables from this list until the abstract counterexamples are eliminated. These approaches provide no guarantees on the size of the abstract model. Our approach, on the other hand, computes the smallest abstract model that can prove the property.

The disadvantage of any refinement-based strategy, is that once some irrelevant constraint is added to the abstract model, it is not removed in subsequent iterations.

As the model checker discovers longer abstract counterexamples, the constraints that were added to eliminate the shorter counterexamples might become redundant. Refinement-based techniques do not identify and remove these constraints from the abstract model. This drawback is present in a refinement-based strategy irrespective of the technique that is used to eliminate spurious counterexamples. The proof-based abstraction technique presented in [Gupta *et al.*, 2003; McMillan and Amla, 2003] tries to alleviate this problem by building a fresh abstract model at each iteration. However, the abstract model is computed from the proof of unsatisfiability produced by SAT-solvers, and this technique is also not optimal. Our approach is not based on refinement. In our iterative loop, the abstract model generated in the next iteration is not necessarily a refinement of the previous abstract model.

Refinement minimization [Wang *et al.*, 2001] is used by various approaches to reduce the size of the abstract model. However, it is an expensive operation, and it can only guarantee *local minimality*, i.e., it only ensures that none of the variables in the abstract abstract model can be removed while still preserving the property. There has been some work in extracting the smallest unsatisfiable subset of a set of clauses [Lynce and Silva, 2004; Oh *et al.*, 2004]. In theory, these techniques can be combined with the proof-based abstraction method [McMillan and Amla, 2003] to generate small abstract models. However, in practice, these techniques can only be applied to instances with a small number of variables and clauses, and therefore do not scale to real-world systems.

5.16 Experimental Results

We implemented a model checker for hardware circuits based on the LearnAbs algorithm. It uses Chaff [Moskewicz *et al.*, 2001] as the SAT-solver, and CPLEX [ILOG] as the ILP-solver. Cadence SMV [McMillan] is used as the BDD-based model checker for verifying the abstract models. We used 3 sets of benchmarks for our experiments: the *IU* benchmarks [Clarke *et al.*, 2004] from Synopsys, Incorporated, the *PJ* benchmarks derived from the PicoJava processor [McMillan and Amla, 2003], and the *RB* benchmarks from the IBM Formal Verification Library [Zarpas, 2004]. The value of N in LearnAbs was set to 25, i.e., we generated 25 broken trace samples in each iteration of LearnAbs. All experiments were performed on a 1.5GHz Dual Athlon machine with 3GB RAM and running Linux. No pre-computed variable ordering files were used in the experiments.

Table 5.1 shows the comparison of LearnAbs with the abstraction strategy based on separating the set of deadend states from the set of bad states [Clarke *et al.*, 2004]. The numbers for *Deadend/Bad* were obtained from [Clarke *et al.*, 2004] (these experiments were performed on the same machine as ours). The table shows the number of latches in the circuit *(reg)*; the counterexample length *(cex)* - *'T'* indicates that the property holds; the total running time *(time)*; the number of model checking calls *(itr)*; and the number of latches in the final abstract model *(abs)*. The LearnAbs algorithm generates smaller abstract models. For smaller benchmarks, model checking is not the bottleneck, and therefore the effort spent in generating a good abstract model does not result in a smaller overall running time.

circuit	reg	cex	Deadend/Bad			LearnAbs		
			time	itr	abs	time	itr	abs
IU30	30	10	6	3	20	8	10	14
IU35	35	19	23	4	21	61	70	16
IU40	40	19	34	5	22	37	35	15
IU45	45	19	39	5	22	34	31	17
IU50	50	19	57	5	22	67	22	16
IU55	55	10	59	3	20	13	10	13
IU60	60	10	77	3	20	34	9	14
IU65	65	10	80	3	20	30	10	13
IU70	70	10	69	3	20	25	10	13
IU75	75	10	23	4	21	22	11	15
IU80	80	10	26	4	21	26	10	14
IU85	85	10	28	4	21	25	9	14
IU90	90	10	28	4	21	21	10	13
IUP1	4494	T	>2hr	-	-	**1295**	18	8

Table 5.1: Comparison of Deadend/Bad with LearnAbs.

The LearnAbs algorithm performs better on the larger benchmarks.

Table 5.2 and Table 5.3 compare the LearnAbs algorithm with the abstraction strategy based on the proof of unsatisfiability generated by SAT-solvers (Satproof) [Chauhan *et al.*, 2002; Gupta *et al.*, 2003; McMillan and Amla, 2003]. We did not use greedy minimization on the unsatisfiable cores in Satproof. For both these techniques, we show results for the single-counterexample mode (indicated by 'S') that eliminates one abstract counterexample in each iteration; and the all-counterexample mode (indicated by 'A') that eliminates all counterexamples at the current depth in each iteration. The LearnAbs algorithm consistently generates smaller abstract models, and this translates to a better or similar runtime on most benchmarks. For the benchmarks with bugs, Satproof performs better because the BMC step is very efficient in identifying the error trace. The LearnAbs algorithm completes all the benchmarks, while Satproof cannot complete 4 benchmarks. On *IUP1*, the SAT-solver runs out of memory while trying to build an unrolling of the concrete model for depth 67 for Satproof (S), and 69 for Satproof (A).

circuit	reg	cex	Satproof (S)			LearnAbs (S)			Satproof (A)			LearnAbs (A)		
			time	itr	abs	time	itr	abs	time	itr	abs	time	itr	abs
PJ00	348	T	7	3	9	6	3	4	7	3	9	6	3	4
PJ01	321	T	6	3	3	5	2	0	6	3	3	5	2	0
PJ02	306	T	7	3	4	6	3	4	7	3	4	6	3	4
PJ03	306	T	6	3	4	5	3	4	6	3	4	6	3	4
PJ04	305	T	7	3	3	5	2	0	6	3	3	5	2	0
PJ05	105	T	9	7	34	35	8	28	9	7	34	34	8	28
PJ06	328	T	209	7	56	23	7	41	210	7	56	24	7	41
PJ07	94	T	18	11	36	13	7	32	20	10	36	14	7	32
PJ08	116	T	58	6	41	75	7	41	58	6	41	76	7	41
PJ09	71	T	8	6	31	4	6	25	8	6	31	4	6	25
PJ10	85	T	3	3	5	2	3	4	3	3	5	2	3	4
PJ11	294	T	11	5	12	5	2	0	12	5	12	5	2	0
PJ12	312	T	4	1	0	4	1	0	4	1	0	4	1	0
PJ13	420	T	10	5	13	8	3	8	10	5	13	8	3	8
PJ14	127	T	25	6	32	9	4	13	35	5	32	9	4	13
PJ15	355	T	184	5	42	14	5	23	185	5	42	15	5	23
PJ16	290	T	248	6	44	64	7	32	246	6	44	65	7	32
PJ17	212	T	2126	14	43	1869	20	29	5037	11	43	3685	14	31
PJ18	145	T	993	22	49	390	8	32	161	7	45	542	7	36
PJ19	52	T	>2hr	-	-	**18**	3	12	>2hr	-	-	**19**	3	12

Table 5.2: Comparison of Satproof and LearnAbs, in *single* (S) and *all* (A) counterexamples mode.

circuit	reg	cex	SATPROOF (S)			LEARNABS (S)			SATPROOF (A)			LEARNABS (A)		
			time	itr	abs	time	itr	abs	time	itr	abs	time	itr	abs
RB05_1	313	31	11	10	24	141	17	13	17	6	12	137	11	13
RB09_1	168	T	1	4	9	1	3	4	1	4	9	1	3	4
RB10_1	236	T	3	5	9	1	4	3	3	5	9	1	4	3
RB10_2	236	T	3	6	11	2	4	3	3	6	11	2	4	3
RB10_3	236	T	5	7	34	2	5	5	5	7	34	1	5	5
RB10_4	236	T	8	10	23	1	4	5	6	8	22	2	4	5
RB10_5	236	T	1	4	7	2	4	4	2	4	7	2	4	4
RB10_6	236	T	3	5	7	2	4	4	2	5	7	2	4	4
RB11_2	242	T	>1hr	-	-	**128**	24	26	>1hr	-	-	**219**	14	30
RB14_1	180	T	37	7	47	3	5	15	37	7	47	3	5	15
RB14_2	180	T	>1hr	-	-	**1258**	60	37	334	11	87	179	13	38
RB15_1	270	9	2	7	8	17	9	4	2	7	8	13	9	4
RB16_1_1	1117	T	8	7	92	324	8	80	8	7	92	260	8	80
RB16_2_4	1113	5	4	5	40	61	5	32	4	5	40	48	5	32
RB26_1	608	T	1	1	0	1	1	0	1	1	0	1	1	0
RB31_2_1	111	T	>1hr	-	-	**38**	27	29	>1hr	-	-	**17**	12	26
IUP1	4494	T	*mem*	-	-	**1295**	18	8	*mem*	-	-	**151**	13	5

Table 5.3: Comparison of SATPROOF and LEARNABS, in *single* (S) and *all* (A) counterexamples mode.

Chapter 6

Complexity of Abstraction

6.1 Smallest Unsatisfiable Cores

Let $C = \{c_1, ..., c_n\}$ be a set of clauses over the set X of boolean variables. Let $w : C \rightarrow Z$ be a weight function that assigns a non-negative integral weight to each clause.

Definition 6.1.1. The *weight* $|R|$ of a set R of clauses, $R \subseteq C$, is defined as the sum of the weights of the clauses in R.

$$|R| = \sum_{c_i \in R} w(c_i)$$

Definition 6.1.2. A set R of clauses, $R \subseteq C$, is said to be satisfiable if the formula F, given by the conjunction of the clauses in R is satisfiable.

$$F = \bigwedge_{c_i \in R} c_i$$

The set R is said to be unsatisfiable if F is unsatisfiable.

Definition 6.1.3. A set R of clauses, $R \subseteq C$, is called a *Smallest Unsatisfiable Core* (SUC) of C if

1. R is unsatisfiable.

2. For all unsatisfiable sets R', $R' \subseteq C$,

$$|R| \leq |R'|$$

6.2 Quantified Boolean Formulas

A *Quantified Boolean Formula* (QBF) is a formula of the form

$$F = Q_0 X_0. \dots Q_d X_d \ \phi$$

where ϕ is a propositional formula over a set X of boolean variables. Without loss of generality, we can assume that ϕ is a conjunction of clauses. The $X_i's$ are mutually disjoint subsets of variables such that $\bigcup_{0 \leq i \leq d} X_i = X$. Each Q_i is either an existential quantifier $\exists$ or a universal quantifier $\forall$, and no two adjacent quantifiers are the same. The depth d is called the *alternation depth* of the QBF formula.

QBF is the standard PSPACE-complete problem, thereby placing it higher in the complexity hierarchy than problems like SAT, which are NP-complete. The class of one-alternation QBF formulas (corresponding to $d = 1$), which we denote by QBF_2, is Σ_2^P-complete. Intuitively, a Σ_2^P-complete problem can be solved in polynomial time by a non-deterministic turing machine that has access to an NP-complete oracle.

6.3 Complexity of SUC

In this section, we show that the decision version of SUC is Σ_2^P-complete. Clearly, this problem lies in Σ_2^P, because it can be solved by a non-deterministic turing machine that guesses a subset of C and checks if the subset is unsatisfiable. To show that the decision version of SUC is Σ_2^P-hard, we describe a translation of QBF_2 to this problem. This translation is similar to the one described in [Umans, 1998].

Theorem 6.3.1. *The problem of checking if a set C of clauses has a SUC of weight k is Σ_2^P-hard.*

Proof. Consider a QBF_2 formula

$$F = \forall X_0.\exists X_1\ \phi(X_0, X_1)$$

such that the set X_0 contains k variables. For each variable $x_i \in X_0$, we introduce two new variables, x_i^+ and x_i^-. The set of all x_i^+ variables is denoted by X_0^+ and the set of all x_i^- variables is denoted by X_0^-. The formula $\psi(X_0^+, X_0^-, X_1)$ is obtained from ϕ by replacing every positive occurrence of x_i with x_i^+, and every negative occurrence of x_i with x_i^-. Let

$$G = (\ \psi(X_0^+, X_0^-, X_1) \vee \bigvee_{x_i \in X_0} (\neg x_i^+ \wedge \neg x_i^-)\ )$$

Let C_G consist of the clauses in the CNF translation of G. Let U consist of $2k$ unit clauses (x_i^+) and (x_i^-), $x_i \in X_0$. Let $C = C_G \cup U$. The clauses in C_G are each assigned a weight 0, while the clauses in U are each assigned a weight 1.

A SUC of C must contain at least one of each pair (x_i^+) and (x_i^-), otherwise the clauses in the SUC can be satisfied by assigning 0 to the missing x_i^+ and x_i^-. Thus, the weight of a SUC of C is at least k. We show that C has a SUC of weight k if and only if the formula F evaluates to 0.

(ONLY IF) Suppose C has a SUC R of weight k. Let $R' = R \cup C_G$. Note that R' is also a SUC of C, since the clauses in C_G have weight 0. Since R' contains at least one of (x_i^+) and (x_i^-) for each $x_i \in X_0$ and each of these clauses has weight 1, the fact that R' has weight k implies that R' contains exactly one of (x_i^+) and (x_i^-) for each $x_i \in X_0$. Consider an assignment to the variables $x_i \in X_0$ obtained by assigning 1 to x_i if $(x_i^+) \in R'$ and 0 to x_i^- if $(x_i^-) \in R'$. This assignment satisfies the clauses in $R' \cap U$. Since R' is a SUC, the clauses in $R' \cap C_G = C_G$ cannot be satisfied after applying this assignment. Thus, we have an assignment to X_0 for which $\phi(X_0, X_1)$ is unsatisfiable, which implies that F evaluates to 0.

(IF) Suppose F evaluates to 0. Then there exists an assignment A to the variables in X_0 for which $\phi(X_0, X_1)$ is unsatisfiable. Let U_1 consist of exactly one of (x_i^+) and (x_i^-) for each $x_i \in X_0$, such that (x_i^+) is in U_1 if x_i is assigned 1 by A, and (x_i^-) is in U_1 if x_i is assigned 0 by A. Let $R = U_1 \cup C_G$. Clearly, $|R| = k$. Also, since $\phi(X_0, X_1)$ is unsatisfiable after applying A, R is unsatisfiable. Thus, C has a SUC of weight k.

Since QBF_2 is Σ_2^P-hard, this proves that the decision version of SUC is Σ_2^P-hard. $\square$

6.4 Refinement and SUC

Consider the separation-based refinement technique described in Section 4.5. This refinement strategy tries to find the smallest set of variables that separates the dead-end and bad states. In this section, we reduce this problem to the task of finding a SUC of a set of clauses.

Let $\mathcal{D}$ (and $\mathcal{B}$) denote the formula for the deadend (and bad) states for the property at hand (Section 4.5). Let $\mathcal{I}$ denote the set of invisible variables. Let

$$F = \mathcal{D} \ \wedge \ \mathcal{B}' \ \wedge \ \bigwedge_{v_i \in \mathcal{I}} (\ d_i \Rightarrow (v_i = v_i') \ )$$

The prime symbol over $\mathcal{B}$ denotes the fact that we replace each variable v_i in $\mathcal{B}$ with a new variable v_i'. The $d_i's$ are also new variables, corresponding to each $x_i \in \mathcal{I}$. Let C_F consist of the clauses in the CNF translation of F. Let U consist of the unit clauses (d_i), for each d_i. Let $C = C_F \cup U$. The clauses in C_F are each assigned a weight 0, while the clauses in U are each assigned a weight 1. Let R be a SUC of C. The smallest separating set for the deadend and bad states consists of the x_i variables corresponding to the (d_i) clauses in $R \cap U$.

6.5 Abstraction and SUC

The aim of abstraction is to find a *small* abstract model such that the property holds on that model (Section 3.2). For localization abstraction, the transition relation R

can be viewed as a conjunction of constraints

$$R(s, s') = R_1(s, s') \wedge R_2(s, s') \ldots \wedge R_n(s, s')$$

For example, for the abstraction framework with visible/invisible variables, each constraint corresponds to the next-state function of a state variable. For fine-grained abstractions, each constraint might correspond to a gate in the circuit. There is a weight associated with each constraint, which is a measure of the increase in the complexity of model checking if that constraint is included in the abstract model. Localization tries to identify a subset of these constraints thats proves the property and has a small weight. Most abstraction frameworks (including LearnAbs, Section 5.6) try to identify the constraints that prove the property for all depths, by identifying constraints that prove the property for some bounded depth. In this section, we show how to reduce the problem of finding the smallest subset of constraints that prove the property up to a given depth, to the task of finding a SUC for a set of clauses.

Consider the formula $R'(s, s', d)$ obtained from R by introducing a *selection variable* d_i for each R_i.

$$R'(s, s', d) = (d_1 \Rightarrow R_1(s, s')) \wedge (d_2 \Rightarrow R_2(s, s')) \ldots \wedge (d_n \Rightarrow R_n(s, s'))$$

Let N^k denote the BMC unfolding of the model (similar to $M^k_{\mathcal{S}}$ and $M^k_{\mathcal{L}}$ in Section 3.5), with R replaced with R'. Let C_N consist of the clauses in the CNF translation of N^k. Let U consist of the unit clauses (d_i), corresponding to each d_i. Let $C = C_N \cup U$. The clauses in C_N are each assigned a weight 0, while the clauses in U are each assigned the weight of the corresponding R_i. Let R be a SUC of C. The smallest abstract model consists of the $R_i's$ corresponding to the clauses in $R \cap U$.

6.6 Duality : SAT vs. UNSAT

This section illustrates the *duality* between satisfiable subsets and unsatisfiable cores. In addition to SUC (Definition 6.1.3), we need the following definitions.

Definition 6.6.1. A set R of clauses, $R \subseteq C$, is called a *Largest Satisfiable Subset* (LSS) of C if

1. R is satisfiable.
2. For all satisfiable sets R', $R' \subseteq C$,

$$|R| \geq |R'|$$

Definition 6.6.2. A set R of clauses, $R \subseteq C$, is called a *Minimal Unsatisfiable Core* (MUC) of C if

1. R is unsatisfiable.
2. Any set R' with $R' \subset R$ is satisfiable.

Definition 6.6.3. A set R of clauses, $R \subseteq C$, is called a *Maximal Satisfiable Subset* (MSS) of C if

1. R is satisfiable.

2. Any set R' with $R \subset R'$ is unsatisfiable.

Algorithm 6.1 computes a MSS of a set of clauses C. Starting with an empty set (line 1), it goes over every clause in C (line 2), and adds it to this set (line 4) if it does not make the set unsatisfiable (line 3). The resulting set is a MSS of C.

Algorithm 6.1 Maximal Satisfiable Subset

```
Mss (C)
1: R = {}; i = 1
2: while (i <= n) do
3:     if (R ∪ {c_i} is satisfiable) then
4:         R = R ∪ {c_i}
5:     i = i + 1
6: return R
```

Algorithm 6.2 computes a MUC of a set of clauses C. Starting with the set of clauses C (line 1), it goes over every clause in C (line 2), and removes it from this set (line 4) if it does not make the set satisfiable (line 3). The resulting set is a MUC of C.

Algorithm 6.3 computes a LSS of a set of clauses C. The output R is initially set to C (line 1). The algorithm terminates when R becomes satisfiable (line 2). At each iteration, it computes an unsatisfiable subset of R (line 3). All these subsets are collected in the set I (line 4). It then computes the smallest cover of the sets in I (line 5) and the set R is assigned the complement of the cover (line 6). A heuristic to speed up the convergence of this algorithm is to compute a MUC (or SUC) of R

Algorithm 6.2 Minimal Unsatisfiable Core

MUC (C)

1: $R = C$; $i = 1$
2: **while** $(i <= n)$ **do**
3:     **if** ($R/\{c_i\}$ is unsatisfiable) **then**
4:         $R = R/\{c_i\}$
5:     $i = i + 1$
6: **return** R

at line 3.

Algorithm 6.3 Largest Satisfiable Subset

LSS (C)

1: $R = C$; $I = \{\}$
2: **while** (R is not satisfiable) **do**
3:     $T =$ any unsatisfiable subset of R
4:     $I = I \cup \{T\}$
5:     $H = MinCover(I)$
6:     $R = C/H$
7: **return** R

Algorithm 6.4 computes a SUC of a set of clauses C. The output R is initially set to the empty set (line 1). The algorithm terminates when R becomes unsatisfiable (line 2). At each iteration, it computes a satisfiable subset of C that contains R (line 3). The complements of these subsets are collected in the set I (line 4). It then computes the smallest cover of the sets in I (line 5) and the set R is assigned this cover (line 6). A heuristic to speed up the convergence of this algorithm is to compute a MSS (or LSS) of C at line 3.

Algorithm 6.4 Smallest Unsatisfiable Core

SUC (C)

1: $R = \{\}; I = \{\}$

2: **while** (R is satisfiable) **do**

3: $T =$ any satisfiable subset of C that contains R

4: $I = I \cup (C/\{T\})$

5: $H = MinCover(I)$

6: $R = H$

7: **return** R

Our LEARNABS algorithm (Chapter 5.6) is very similar to SUC (Algorithm 6.4). The broken traces essentially correspond to satisfiable subsets of the set of clauses for the BMC unfolding of the property, and our good sampling heuristic tries to speed up convergence by generating maximal (or largest) satisfiable subsets of this set of clauses.

Chapter 7

Abstraction-Refinement for Bounded Model Checking

7.1 Preliminaries

7.1.1 Bounded Model Checking

SAT-based Bounded Model Checking (BMC) (Section 3.5) is a powerful technique for refuting properties. Given a model M, a property φ and a positive integer k representing the depth of the search, a Bounded Model Checker generates a propositional formula that is satisfiable if and only if there is a counterexample of length k or less to φ, in M. In this case we write $M \not\models_k \varphi$. The idea is to iteratively deepen the search for counterexamples until either a bug is found or the problem becomes too hard to solve in a given time limit. There is a very weak correlation, if any, between what is hard for standard BDD-based model checking, and BMC. It should be clear, then, that every attempt to make this technique work faster, and hence enable to check larger circuits and in deeper cycles, is worth while.

7.1.2 Unsatisfiable cores generated by SAT-solvers

While a satisfying assignment is a checkable proof that a given propositional formula is satisfiable, until recently SAT-solvers produced no equivalent evidence when the formula is unsatisfiable. The notion of generating resolution proofs from a SAT-solver was introduced in [McMillan and Amla, 2003]. From this resolution proof, one may also extract the *unsatisfiable core*, which is the set of clauses from the original CNF formula that participate in the proof. Topologically, these are the roots of the resolution graph. The importance of the unsatisfiable core is that it represents a subset, hopefully a small one, of the original set of clauses that is unsatisfiable by itself. This information can be valuable in an abstraction-refinement process as well as in other techniques, because it can point to the reasons for unsatisfiability. In the case of abstraction-refinement, it can guide the refinement process, since it points to the reasons for why a given spurious counterexample cannot be satisfied together with the concrete model.

7.1.3 Counterexample-Guided Abstraction-Refinement

Given a model M and a safety or liveness property φ, the abstraction-refinement framework (Section 4.1) encapsulates various automatic algorithms for finding an abstract model $\hat{M}$ with the following two properties:

- $\hat{M}$ over-approximates M, and therefore $\hat{M} \models \varphi \rightarrow M \models \varphi$;
- $\hat{M}$ is smaller than M, so checking whether $\hat{M} \models \varphi$ can be done more efficiently than checking the original model M.

This framework is an important tool for tackling the state-explosion problem in model checking. Algorithm 7.1 describes a particular implementation of the Counterexample-Guided Abstraction-Refinement (CEGAR) loop. We denote by $BMC(M, \varphi, k)$ the process of generating the length k BMC unfolding for model M and solving it, according to the standard BMC framework as explained in Section 3.5. The loop simulates the counterexample on the concrete model using a SAT-solver (line 4), and uses the unsatisfiable core produced by the SAT-solver to refine the abstract model (lines 5,6). This refinement strategy was proposed by Chauhan et al.[Clarke *et al.*, 2002].

7.2 Abstraction-Refinement for Bounded Model Checking

The underlying principles behind the CEGAR framework are the following:

- The information that was used to eliminate previous counterexamples, which is captured by the abstract model, is relevant for proving the property.

Algorithm 7.1 Counterexample-Guided Abstraction-Refinement

CEGAR (M, φ)

1: $\hat{M} = \{\}$

2: **if** $MC(\hat{M}, \varphi) = TRUE$ **then return** 'TRUE'

3: **else** let C be the length k counterexample produced by the model checker

4: **if** $BMC(M, k, \varphi) \wedge C = SAT$ **then return** 'bug found in cycle k'

5: **else** let U be the set of gates in the unsatisfiable core produced by the SAT-solver

6: $\hat{M} = \hat{M} \cup U$

7: **goto** line 2

- If the abstract model does not prove the property, then the counterexamples in the abstract model can guide the search for a refinement.

We apply these principles to guide the SAT-solver, thereby making BMC faster. The underlying assumption of our technique is that the information necessary for proving the property along the various cycles is relatively stable. In other words, similar abstract models can verify the property along consecutive cycles. This assumption is exactly what explains the success of CEGAR. We use the same assumption, but for making BMC itself work faster.

7.2.1 The CG-BMC algorithm

The pseudo-code of Counterexample-Guided Bounded Model Checking (CG-BMC) is shown in Algorithm 7.2. We start with an empty initial abstraction and an initial search depth $k = 1$. In each iteration of the CG-BMC loop, we first try to find a counterexample in the abstract model (line 3). If there is no counterexample in the abstract model, the property holds at cycle k and the abstract model now contains the gates in the unsatisfiable core generated by the SAT-solver (lines 4,5). Otherwise, if a counterexample is found, we simulate the counterexample on the concrete model (line 8). If the counterexample can be concretized, we report a real bug. If the counterexample is spurious, the abstract model is refined by adding the gates in the unsatisfiable core (line 10). Like standard BMC, CG-BMC either finds an error or continues until it becomes too complex to solve within a given time limit. It can be combined, like standard BMC, with techniques to make it complete, such as finding the Completeness Threshold (an upper bound on the length of the shortest counterexample if one exists) [Kroening and Strichman, 2003].

7.2.2 Inside a SAT-Solver

Most modern SAT-solvers are based on the DPLL search procedure [Davis and Putnam, 1960]. The search for a satisfying assignment in the DPLL framework is organized as a binary search tree in which at each level a *decision* is made on the variable to split on, and the first branch to be explored (each of the two branches corresponds to a different Boolean assignment to the chosen variable). After each decision, Boolean Constrain Propagation (BCP) is invoked, a process that finds the implications of the last decision by iteratively applying the *unit-clause rule* (the unit-clause rule simply says that if in an l-length clause $l-1$ literals are unsatisfied, then the last literal must be satisfied in order to satisfy the formula). Most of the

Algorithm 7.2 Counterexample-Guided Bounded Model Checking

CG-BMC (M, φ)

1: $k = 0$; $\hat{M} = \{\}$
2: $k = k + 1$
3: **if** $BMC(\hat{M}, k, \varphi) = UNSAT$ **then**
4: Let U be the set of gates in the unsatisfiable core produced by the SAT-solver
5: $\hat{M} = U$
6: **goto** line 2
7: **else** let C be the satisfying assignment produced by the SAT-solver
8: **if** $BMC(M, k, \varphi) \wedge C = SAT$ **then return** 'bug found in cycle k'
9: **else** let U be the set of gates in the unsatisfiable core produced by the SAT-solver
10: $\hat{M} = \hat{M} \cup U$
11: **goto** line 3

computation time inside a SAT-solver is spent on BCP. If BCP leads to a conflict (an empty clause), the SAT-solver backtracks and changes some previous decision.

The performance of a SAT-solver is determined by the choice of the decision variables. Typically SAT-solvers compute a score function for each undecided variable that prioritizes the decision options. Many branching heuristics have been proposed in the literature: see, for example, [Silva, 1999]. The basic idea behind many of these heuristics is to increase the score of variables that are involved in conflicts, thereby moving them up in the decision order. This can be viewed as a form of refinement [McMillan, 2003].

An obvious question that comes to mind is why do we need an abstraction-refinement framework for BMC when a SAT-solver internally behaves like a refinement engine. A major drawback of the branching heuristics in a SAT-solver is that they have no global perspective of the structure of the problem. While operating on a BMC instance, they tend to get 'distracted' by local conflicts that are not relevant to the property at hand. CG-BMC avoids this problem by forcing the SAT-solver to find a satisfying assignment to the abstract model, which contains only the relevant part of the concrete model. It involves the other variables and gates only if it is not able to prove unsatisfiability with the current abstract model.

The method suggested by Wang et al. [Wang *et al.*, 2004b] that we describe in the related work section (Section 7.3), tries to achieve a similar effect by modifying the branching heuristics. They perform BMC on the concrete model, while changing the score function of the SAT-solver so it gives higher priority to the variables in the abstract model (they do not explicitly refer to an abstract model, rather to the unsatisfiable core of the previous iteration, which is what we refer to as the abstract model). Since their SAT-solver operates on a much larger concrete model, it spends a lot more time doing BCP. Moreover, many of the variables in the abstract model are also present in clauses that are not part of the abstract model and their method often encounters conflicts on these clauses. CG-BMC, on the other hand, *isolates* the abstract model and solves it separately, in order to avoid this problem.

7.2.3 The CG-BMC-T algorithm

The following is an implicit assumption in the CG-BMC algorithm: Given two unsatisfiable sets of clauses C_1 and C_2 such that $C_1 \subset C_2$, solving C_1 is faster than solving C_2. While this is a reasonable assumption and mostly holds in practice, it is not always true. It is possible that the set of clauses C_2 is *over-constrained*, so that the SAT-solver can prove its unsatisfiability with a small search tree. Removing clauses from C_2, on the other hand, could produce a set of clauses C_1 that is no longer over-constrained and proving the unsatisfiability of C_1 could take much more time [Crawford and Anton, 1993].

We observed this phenomenon in some of our benchmarks. As an example, consider circuit *PJ05* in Table 7.1 (see Section 7.4). The CG-BMC algorithm takes much longer than BMC to prove the property on *PJ05*. This is not because of the overhead of the refinement iterations: the abstract model has enough clauses to prove the property after an unfolding length of 9. The reason for this is that BMC on the small abstract model takes more time than BMC on the original model.

In order to deal with such situations, we propose a modified CG-BMC, called CG-BMC-T (T stands for Timeout). Algorithm 7.3 describes the pseudo-code for CG-BMC-T. The intuition behind this algorithm is the following: if the SAT-solver is taking a long time on the abstract model, we check if it can quickly prune away this search region by adding some constraints from the concrete model. In each iteration of the CG-BMC-T loop, we set a timeout T for the SAT-solver (line 3) and try to find a counterexample in the abstract model (line 4). If the SAT-solver completes, the loop proceeds like CG-BMC. However, if the SAT-solver times-out, we simulate the partial assignment on the concrete model with a smaller timeout ($T \times \beta,\ \beta < 1$) (lines 13,15). If the concrete model is able to concretize the partial assignment, we report a bug (line 16). If the concrete model refutes the partial assignment, we add the unsatisfiable core generated by the SAT-solver to the abstract model, thereby eliminating the partial assignment (line 20). If the concrete solver also times-out, we go back to the abstract solver. However, for this next iteration, we increase the timeout with a factor of α (line 13). The CG-BMC-T algorithm is more robust, as indicated by our experiments.

7.3 Related Work

An alternative to our two-stage heuristic, corresponding to checking the abstract and concrete models, is to try to emulate this process within a SAT-solver by controlling the decision heuristic (focusing first on the parts of the model corresponding to the

Algorithm 7.3 CG-BMC with Timeouts

CG-BMC-T (M, φ)

1: $k = 0$; $\hat{M} = \{\}$

2: $k = k + 1$; $T = T_{init}$

3: $Set\_Timeout(T)$

4: $Res = BMC(\hat{M}, k, \varphi)$

5: **if** $Res = UNSAT$ **then**

6:     Let U be the set of gates in the unsatisfiable core produced by the SAT-solver

7:     $\hat{M} = U$

8:     **goto** line 2

9: **else**

10:     **if** $Res = SAT$ **then**

11:         Let C be the satisfying assignment produced by the SAT-solver

12:     **else**

13:         $T = T \times \alpha$; $Set\_Timeout(T \times \beta)$

14:         Let C be the partial assignment produced by the SAT-solver

15: $Res = BMC(M, k, \varphi) \wedge C$

16: **if** $Res = SAT$ **then return** 'bug found in cycle k'

17: **else**

18:     **if** $Res = UNSAT$ **then**

19:         Let U be the set of gates in the unsat core produced by the SAT-solver

20:         $\hat{M} = \hat{M} \cup U$

21: **goto** line 3

abstract model). Wang et al. [Wang *et al.*, 2004b] went in this direction: they use the unsatisfiable cores from previous cycles to guide the search of the SAT-solver when searching for a bug in the current cycle. Guidance is done by changing the variable selection heuristic to first decide on the variables that participated in the previous unsatisfiable cores. Furthermore, McMillan observed in [McMillan, 2003] that modern SAT-solvers internally behave like abstraction-refinement engines by themselves: their variable selection heuristics move variables involved in recent conflicts up in the decision order. However, a major drawback of this approach is that while operating on a BMC instance, they propagate values to variables that are not part of the abstract model that we wish to concentrate on, and this can lead to long phases in which the SAT-solver attempts to solve *local conflicts* that are irrelevant to proving the property. In other words, this approach allows irrelevant clauses to be pulled into the proof through propagation and cause conflicts. This is also the drawback of [Wang *et al.*, 2004b], as we will prove by experiments. Our approach solves this problem by forcing the SAT-solver to first find a complete abstract trace before attempting to refute it. We achieve this by isolating the important clauses from the rest of the formula in a separate SAT instance.

Another relevant work is by Gupta et al. [Gupta *et al.*, 2003], that presents a top-down abstraction framework where proof analysis is applied iteratively to generate successively smaller abstract models. Their work is related because they also suggest using the abstract models to perform deeper searches with BMC. However, their overall approach is very different from ours.

We take the CG-BMC approach one step further in Section 7.5.2, by considering a new abstraction-refinement framework, in which CG-BMC unifies CEGAR and Proof-Based Refinement (PBR) [Gupta *et al.*, 2003; McMillan and Amla, 2003]. PBR eliminates all counterexamples of a given length in a single refinement step. The PBR refinement uses the unsatisfiable core of the (unrestricted) BMC instance to generate a refinement. Amla et al. showed in [Amla and McMillan, 2004] that CEGAR and PBR are two extreme approaches: CEGAR burdens the model checker by increasing the number of refinement iterations while PBR burdens the refinement step because the BMC unfolding without the counterexample constraints is harder to refute. They also present a hybrid approach that tries to balance between the two, which results in a more robust overall behavior. We show that by replacing BMC with a more efficient CG-BMC as the refinement engine inside PBR, we also get a hybrid abstraction-refinement framework that can balance the model checking and refinement efforts.

7.4 Experimental Results

We implemented our techniques on top of the SAT-solver Chaff [Moskewicz *et al.*, 2001]. Some modifications were made to Chaff to produce unsatisfiable cores while adding and deleting clauses incrementally. Our experiments were conducted on a set of benchmarks that were derived during the formal verification of an open source Sun PicoJava II microprocessor [McMillan and Amla, 2003]. All experiments were performed on a 1.5GHz Dual Athlon machine with 3Gb RAM. We set a timeout of 2 hours and a maximum BMC search depth of 60.

We use the incremental feature of Chaff to optimize the CG-BMC loop as follows. We maintain two incremental SAT-instances: *solver-Abs* contains the BMC unfolding of the abstract model while *solver-Conc* contains the BMC unfolding of the concrete model. The counterexample generated by *solver-Abs* is simulated on *solver-Conc* by adding unit clauses. The unsatisfiable core generated by *solver-Conc* is added to *solver-Abs*. Our algorithm can in principle be implemented inside a SAT-solver although this requires fundamental changes in the way it works, and it is not clear if it will actually perform better or worse. We discuss this option further in Chapter 8.

Table 7.1 compares our techniques with standard BMC. For each circuit, we report the depth that was completed by all techniques, the runtime in seconds, and the number of backtracks (for CG-BMC/CG-BMC-T we report the backtracks on both abstract and concrete models). We see a significant overall reduction in runtime. This reduction is due to a decrease in the total number of backtracks, and the fact that most of the backtracks (and BCP) are performed on a much smaller abstract model. We also observe a more robust behavior with CG-BMC-T ($T_{init} = 10\text{s}, \alpha = 1.5, \beta = 0.2$).

Table 7.2 compares our technique with the approach based on modifying the SAT-solver's branching heuristics, as described in Wang et al. [Wang *et al.*, 2004b]. We report results for both *static* (Ord-Sta) and *dynamic* (Ord-Dyn) ordering methods. The *static* ordering method gives preference to variables in the abstract model throughout the SAT-solving process. The *dynamic* ordering method switches to the SAT-solver's default heuristic after a threshold number of decisions. Our approach

Circuit	Depth	Time(s)			Backtracks				
		BMC	CG-BMC	CG-BMC-T	BMC	CG-BMC		CG-BMC-T	
						Abs	Conc	Abs	Conc
PJ00	35	7020	48	48	139104	378	27	378	27
PJ01	60	273	99	99	169	187	9	187	9
PJ02	39	6817	51	51	79531	807	5	807	5
PJ03	39	6847	51	51	79531	807	5	807	5
PJ04	60	125	99	98	169	184	7	184	7
PJ05	25	751	2812	296	12476	582069	59	64846	445
PJ06	33	2287	2421	364	23110	346150	92	82734	137
PJ07	60	1837	789	449	34064	197843	86	111775	132
PJ08	60	5061	201	201	43564	44468	124	44468	124
PJ09	60	1092	110	110	22858	32453	57	32453	57
PJ10	50	6696	47	46	76153	3285	67	3285	67
PJ11	33	6142	69	70	120158	1484	95	1484	95
PJ12	24	5266	28	28	117420	2029	91	2029	91
PJ13	60	327	103	102	1005	4019	4	4019	4
PJ14	60	5086	295	316	103217	64392	84	62944	93
PJ15	34	6461	117	115	86567	16105	111	16105	111
PJ16	56	4303	172	173	37843	30528	56	30528	56
PJ17	20	7039	815	1153	81326	68728	548	72202	2530
PJ18	43	7197	719	992	170988	102186	1615	126155	1904
PJ19	9	5105	2224	2555	544941	522702	2534	522460	34324
AVG		**4286**	**563**	**365**					

Table 7.1: Comparison of CG-BMC/CG-BMC-T with standard BMC.

performs better than these methods.

7.5 A hybrid approach to refinement

7.5.1 Proof-based Refinement and a hybrid approach

We described the CEGAR loop in Section 7.1.3. An alternative approach, called Proof-Based Refinement (PBR), was proposed by McMillan et al. [McMillan and Amla, 2003] (and independently by Gupta et al.[Gupta *et al.*, 2003]). The pseudo-code for this approach is shown in Algorithm 7.4. In each refinement iteration, PBR performs BMC on the concrete model (line 4) and uses the unsatisfiable core as the abstract model for the next iteration (line 6). As opposed to CEGAR that eliminates one counterexample, PBR eliminates all counterexamples of a given length in a single refinement step.

Algorithm 7.4 Proof-Based Refinement

PBR(M, φ)

1: $\hat{M} = \{\}$

2: **if** $MC(\hat{M}, \varphi) = TRUE$ **then return** 'TRUE'

3: **else** let k be the length of the counterexample produced by the model checker

4: **if** $BMC(M, k, \varphi) = SAT$ **then return** 'bug found in cycle k'

5: **else** let U be the set of gates in the unsatisfiable core produced by the SAT-solver

6: $\hat{M} = U$

7: **goto** line 2

Circuit	Depth	Time(s)			
		Ord-Sta	Ord-Dyn	CG-BMC	CG-BMC-T
PJ00	60	961	942	104	104
PJ01	60	729	711	99	99
PJ02	60	694	678	101	101
PJ03	60	693	679	101	100
PJ04	60	656	641	99	98
PJ05	25	219	205	2812	296
PJ06	20	4786	1192	148	154
PJ07	25	4761	124	40	41
PJ08	60	703	712	201	201
PJ09	60	494	483	110	110
PJ10	54	827	6493	54	69
PJ11	60	816	796	135	111
PJ12	60	1229	973	101	101
PJ13	60	673	657	103	102
PJ14	60	3101	2746	295	316
PJ15	60	3488	3456	296	371
PJ16	60	3022	3021	198	199
PJ17	21	3132	6114	1069	1570
PJ18	38	6850	4846	556	721
PJ19	5	5623	176	113	116
AVG		**1829**	**1077**	**136**	**117**

Table 7.2: Comparison of CG-BMC/CG-BMC-T with Wang et al. [Wang *et al.*, 2004b].

Amla et. al. [Amla and McMillan, 2004] performed an industrial evaluation of the two approaches and concluded that PBR and CEGAR are extreme approaches. PBR has a more expensive refinement step than CEGAR, since PBR performs unrestricted BMC while CEGAR restricts BMC to the counterexample produced by the model checker. CEGAR, on the other hand, has a larger number of refinement iterations since it only eliminates one counterexample per refinement iteration, thereby putting more burden on the model checker. To balance the two, they propose a hybrid of the two approaches.

Their hybrid approach also performs BMC on the concrete model after given a counterexample from the abstract model. However, they use the counterexample only to provide the initial decisions to the SAT-solver. They also set a time limit to the SAT-solver. If the SAT-solver completes before the time-out with an UNSAT answer, the hybrid approach behaves like PBR. On the other hand if the SAT-solver times-out, they rerun a BMC instance conjoined with constraints on some of the variables in the counterexample (but not all). From this instance they extract an unsatisfiable core, thereby refuting a much larger space of counterexamples (note that this instance has to be unsatisfiable if enough time was given to the first instance due to the initial decisions). Their experiments show that the hybrid approach is more robust than PBR and CEGAR.

7.5.2 A hybrid approach based on CG-BMC

Since the CG-BMC algorithm outperforms BMC, it can be used as a replacement for BMC in the refinement step of PBR. For example, in our experiments on *PJ17* (see Section 7.4), CG-BMC proved the property up to a depth of 29, and model checking could prove the correctness of the property on the generated abstract model. BMC could only finish up to depth of 20, and the resulting abstract model had a spurious counterexample of length 21.

CG-BMC is also a better choice than BMC because it provides an elegant way of balancing the effort between model checking and refinement. Algorithm 7.5 shows the pseudo-code of HYBRID, that we obtained after replacing BMC with CG-BMC inside the refinement step of PBR, and adding a choice function (line 2). At each iteration, HYBRID chooses either the model checker (line 3) or a SAT-solver (line 8) to find a counterexample to the current abstract model. The model checker returns 'TRUE' if the property holds for all cycles (line 3). If the SAT-solver returns UNSAT, the property holds at the current depth and the unsatisfiable core is used as the abstract model for the next iteration (line 10). If a counterexample is produced by either the model checker or the SAT-solver, it is simulated on the concrete model (line 13). If the counterexample is spurious, the unsatisfiable core generated by the SAT-solver is added to the abstract model (line 15).

At a first glance, the HYBRID algorithm looks like a CEGAR loop, with the additional option of using a SAT-solver instead of a model checker for verifying the abstract model. However, the HYBRID algorithm captures both the CEGAR and the

PBR approaches. If the choice function always chooses the model checker (line 3), it corresponds to the standard CEGAR algorithm. Now consider a strategy that chooses the model checker every time there is an increase in k at line 10, but chooses the SAT-solver (line 8) in all other cases. This is exactly the PBR loop that uses CG-BMC instead of BMC. Other choice functions correspond to a hybrid approach. There can be many strategies to make this choice, some of which are:

1. Use previous runtime statistics to decide which engine is likely to perform better on the next model. Occasionally switch to give the other engine a chance.

2. Measure the *stability* of the abstract model. That is, whether in the last few iterations (increases of k) there was a need for refinement. Only if not - send it to a model checker.

3. Run the two engines in parallel. If the model checker completes with a counterexample, set k to the cycle number it reached, terminate the SAT process, and continue. If it proved correctness then exit. If the SAT-solver finds a counterexample first - terminate the model checker and continue.

Algorithm 7.5 Hybrid of CEGAR and PBR

HYBRID (M, φ)

```
1:  k = 1; M̂ = {}
2:  goto line 3 OR goto line 8
3:  if MC(M̂, φ) = TRUE then return 'TRUE'
4:  else
5:      Let C be the length r counterexample produced by the model checker
6:      k = r
7:      goto line 13
8:  if BMC(M̂, k, φ) = UNSAT then
9:      Let U be the set of gates in the unsatisfiable core produced by the SAT-solver
10:     M̂ = U; k = k + 1
11:     goto line 2
12: else let C be the satisfying assignment produced by the SAT-solver
13: if BMC(M, k, φ) ∧ C = SAT then return 'bug found in cycle k'
14: else let U be the set of gates in the unsatisfiable core produced by the SAT-solver
15: M̂ = M̂ ∪ U
16: goto line 2
```

Chapter 8

Conclusions and Future Work

8.1 Thesis Contributions

The following are the major contributions of this thesis:

1. We present an application of SAT and machine learning techniques to the counterexample-guided abstraction-refinement framework. Our algorithm outperforms standard model checking, both in terms of execution times and memory requirements. Compared to the state-of-the-art refinement strategy based on the proof of unsatisfiability produced by SAT-solvers [Chauhan *et al.*, 2002], our approach performs better on several benchmarks. This work has been published in [Clarke *et al.*, 2002, 2004].

2. Most abstraction techniques in literature are based on refinement. We formulate abstraction as an inductive learner that learns the abstract model from

samples of broken traces. We present an iterative algorithm for abstraction-based model checking that is not based on refinement, and that can generate the smallest abstract model that proves the property. We evaluate our algorithm on a large set of industrial circuits, and compare it against our learning-based CEGAR approach [Clarke *et al.*, 2002, 2004] and the proof-based abstraction techniques [Chauhan *et al.*, 2002; McMillan and Amla, 2003]. The results indicate that our technique generates much smaller abstract models, leading to better overall performance and a more robust behavior on harder benchmarks. This work has been published in [Gupta and Clarke, 2005].

3. Most previous work has focused on applying CEGAR to model checking. We present a CEGAR technique for BMC. Our technique makes BMC much faster, as indicated by our experiments. BMC is also used for generating refinements in the PBR framework. We show that our technique unifies PBR and CEGAR into an abstraction-refinement framework that can balance the model checking and refinement efforts. This work has been published in [Gupta and Strichman, 2005].

8.2 Future Directions

8.2.1 Learning Abstractions

The work in this thesis has focused on localization abstraction (Section 3.3.2). Localization abstraction will work only if the property at hand is *localizable*, i.e., its

```
int main () {
     int x, y;
1:   x = 100;
2:   y = 100;
3:   while (x ≠ 0) {
4:      x --;
5:      y --;
     }
6:  assert(y == 0);
}
```

Figure 8.1: An example C program.

validity depends on a small part of the system description. The techniques presented in this thesis can also be used with predicate abstraction (Section 3.3.1), which does not suffer from this drawback. Using an example, we provide some intuition on the issues involved in combining our techniques with predicate abstraction.

Consider the C program in Figure 8.1. A concrete state for this program is an assignment to $\{PC, x, y\}$, where PC (program counter) indicates the location in the program and x and y are the program variables. We want to show that the assertion holds on the program, i.e. the state $(6, \_, y \neq 0)$ is not reachable. We construct an initial abstract model for the system using the predicates $b_1 : (x = 0)$ and $b_2 : (y = 0)$ that occur syntactically in the C program. The abstract model tracks the values of these predicates, instead of the variables x and y. Thus, an abstract state is an assignment to $\{PC, b_1, b_2\}$ where b_1 and b_2 are boolean variables corresponding to the predicates. Model checking of this abstract model produces the counterexample in Figure 8.2. This counterexample is spurious, which indicates that the current set of predicates are not sufficient to prove the property and new predicates need to be generated.

Figure 8.3 shows a broken trace corresponding to this counterexample. This broken trace breaks at cycle 4. An abstraction function with predicate $(x = y)$ eliminates this broken trace. The predicates $(x = 1)$ and $(x = 100)$ also eliminate this broken trace. However, the predicate $(x = y)$ is a better choice because it proves the property, while the other eliminating predicates generate longer spurious counterexamples.

The key to combining LearnAbs with predicate abstraction is an efficient way of computing these eliminating predicates from the set of broken trace samples. Many techniques have been proposed in the machine learning community to compute classifiers for a set of sample points [Mitchell, 1997]. These techniques can be applied to the broken trace samples to infer the eliminating predicates.

8.2.2 Abstraction-Refinement for BMC

There are many directions in which our research on abstraction-refinement for BMC can go further. First, hybrid (Algorithm 7.5) should be evaluated empirically, and appropriate choice functions should be devised. The three options we listed

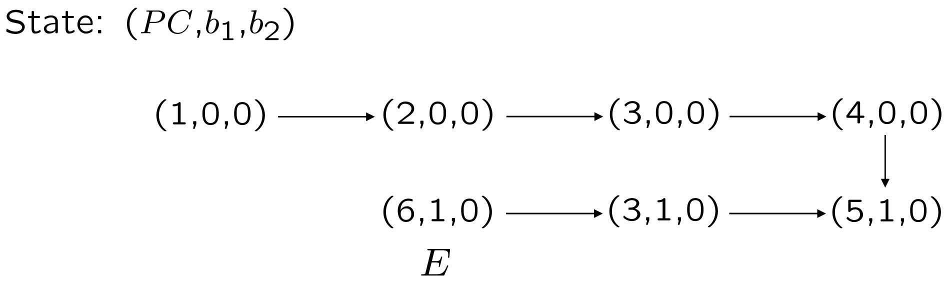

Figure 8.2: Counterexample on the abstract model of the C program in Figure 8.1 with predicates $b_1 : (x = 0)$ and $b_2 : (y = 0)$.

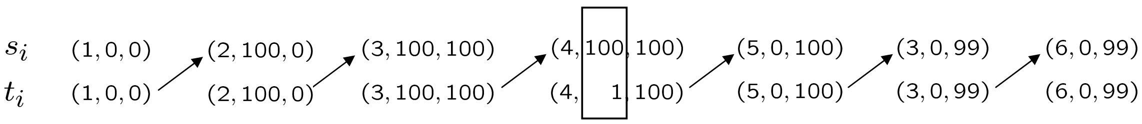

Figure 8.3: Broken trace corresponding to the counterexample in Figure 8.2. Each state is an assignment to $\{PC, x, y\}$.

in Section 7.5.2 are probably still naive. Based on the experiments of Amla et al. reported in [Amla and McMillan, 2004] in hybrid approaches, it seems that this can provide a better balance between the efforts spent in model checking and refinement, and also enjoy the benefit of CG-BMC (Algorithm 7.2). Second, it is interesting to check whether implementing CG-BMC inside a SAT-solver can make it work faster. This requires significant changes in various fundamental routines in the SAT-solver. In particular, this requires some mechanism for clustering the clauses inside the SAT-solver into abstraction levels, and postponing BCP on the clauses in the lower levels until all the higher levels are satisfied. Refinement would correspond to moving clauses up (and down) across levels, possibly based on their involvements in conflicts. A third direction for future research is to explore the application of CG-BMC to other theories and decision procedures, like bit-vector arithmetic.

Bibliography

Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. PBS: A Backtrack-Search Psuedo-Boolean Solver and Optimizer. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

Nina Amla and Ken McMillan. A hybrid of counterexample-based and proof-based abstraction. In *Formal Methods in Computer-Aided Design, 5th International Confrence, FMCAD 2004*, pages 260–274, 2004.

Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

Felice Balarin and Alberto Sangiovanni-Vincentelli. An iterative approach to language containment. In C. Courcoubetis, editor, *Proc. 5^{th} Intl. Conference on Computer Aided Verification (CAV'94)*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 29–40. Springer-Verlag, 1993.

Sharon Barner, Daniel Geist, and Anna Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *Proc. of International Conference on Computer-Aided Verification (CAV)*, Copenhagen, Denmark, July 2002.

Michel Berkelaar. lpsolve, version 2.0. Eindhoven Technical University, The Netherlands.

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *In Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS, pages 193–207, Amsterdam, The Netherlands, March 1999.

Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

Pankaj Chauhan, Edmund Clarke, Somesh Jha, James Kukula, Tom Shiple, Helmut Veith, and Dong Wang. Non-linear quantification scheduling for efficient image computation. In *Proceedings of International Conference on Computer-Aided Design (ICCAD'01)*, pages 293–298, 2001.

Pankaj Chauhan, Edmund Clarke, Somesh Jha, James Kukula, Helmut Veith, and Dong Wang. Using combinatorial optimization algorithms for efficient image computation. In *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, pages 293–309, 2001.

Pankaj Chauhan, Edmund 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 *Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, LNCS, 2002.

Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. *Int. Journal of Software Tools for Technology Transfer (STTT)*, 2(4):410–425, 2000.

Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, pages 359–364, Copenhagen, Denmark, July 2002.

Edmund Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comp. Sci.*, pages 52–71. Springer-Verlag, 1981.

Edmund Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

Edmund Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E.A. Emerson and A.P. Sistla, editors, *Proc.* 12^{th} *Intl. Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2000.

Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc. 14^{th} Intl. Conference on Computer Aided Verification (CAV'02)*, LNCS, pages 265–279. Springer-Verlag, 2002.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, pages 752–794, 2003.

Edmund Clarke, Anubhav Gupta, and Ofer Strichman. SAT based counterexample-guided abstraction-refinement. *IEEE Transactions on Computer Aided Design (TCAD)*, 23(7):1113–1123, 2004.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, 1977.

James M. Crawford and Larry D. Anton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27. AAAI Press, 1993.

Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, 2001. June 2001, Boston, USA.

Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In O'Leary and Aagaard, editors, *Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, LNCS, Portland, Oregon, Nov 2002.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

Marcelo Glusman, Gila Kamhi, Sela Mador-Haim, and Ranan Fraer andMoshe Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003,*, LNCS, pages 176–191, 2003.

Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, Haifa, Israel, June 1997.

Anubhav Gupta and Edmund Clarke. Reconsidering CEGAR: Generating good abstractions without refinement. In *International Conference on Computer Design (ICCD'05)*, pages 591–598, 2005.

Anubhav Gupta and Ofer Strichman. Abstraction-refinement for bounded model checking. In *Proc.* 17^{th} *Intl. Conference on Computer Aided Verification (CAV'05)*, LNCS, pages 112–124. Springer-Verlag, 2005.

Aarti Gupta, Malay K. Ganai, Zijiang Yang, and Pranav Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *International Conference on Computer-Aided Design (ICCAD'03)*, pages 416–423, 2003.

ILOG. CPLEX. www.cplex.com.

Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Word level predicate abstraction and refinement for verifying RTL verilog. In *Proceedings of the 42nd Annual Conference on Design Automation (DAC'05)*, 2005.

Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Proc.* 4^{th} *Intl. Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309, NYU, New-York, January 2003. Springer Verlag.

Robert Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata- Theoretic Approach*. Princeton University Press, 1995.

Jorn Lind-Nielsen and Henrik Reif Andersen. Stepwise CTL model checking of state/event systems. In N. Halbwachs and D. Peled, editors, *Proc.* 11^{th} *Intl. Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 316–327. Springer-Verlag, 1999.

Yuan Lu. *Automatic Abstraction in Model Checking*. PhD thesis, Carnegie Mellon University, 2000.

Ines Lynce and Joao Marques Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.

Freddy Y.C. Mang and Pei-Hsin Ho. Abstraction refinement by controllability and cooperativeness analysis. In *Proceedings of the 41st Annual Conference on Design Automation (DAC'04)*, pages 224–229, 2004.

Ken McMillan. Cadence SMV. Cadence Berkeley Labs, CA.

Ken McMillan and Nina Amla. Automatic abstraction without counterexamples. In *9th Intl. Conf. on Tools And Algorithms For The Construction And Analysis Of Systems (TACAS'03)*, volume 2619 of *LNCS*, 2003.

Ken McMillan. From bounded to unbounded model checking, 2003.

Tom M. Mitchell. *Machine Learning*. WCB/McGraw-Hill, 1997.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference 2001 (DAC'01)*, 2001.

Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 518–523, 2004.

J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1986.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

Joao Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *9th Portuguese Conference on Artificial Intelligence (EPIA)*, 1999.

Ming Tan and Jeffery C. Schlimmer. Two case studies in cost-sensitive concept acquisition. In *AAAI-90: Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

Christopher Umans. The minimum equivalent dnf problem and shortest implicants. In *FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, 1998.

Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal property verification by abstraction-refinement with formal, simulation and hybrid engines. In *Proceedings of Design Automation Conference 2001 (DAC'01)*, 2001.

Chao Wang, Bing Li, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. Improving ariadne's bundle by following multiple threads in abstraction refinement. In *International Conference on Computer-Aided Design (ICCAD'03)*, pages 408–415, 2003.

Chao Wang, Gary D. Hachtel, and Fabio Somenzi. Fine-grain abstraction and sequential don't cares for large scale model checking. In *International Conference on Computer-Aided Design (ICCAD'04)*, pages 112–118, 2004.

Chao Wang, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. Refining the SAT decision ordering for bounded model checking. In *ACM/IEEE 41th Design Automation Conference (DAC'04)*, pages 535–538, 2004.

Emmanuel Zarpas. Simple yet efficient improvements of SAT based bounded model checking. In *Sixth International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, LNCS, 2004.