# Automatic Abstraction in Model Checking

**Yuan Lu**

December 2000

Department of Electrical and Computer Engineering
Carnegie Institute of Technology
Carnegie Mellon University
Pittsburgh, PA 15213

This research is sponsored by the Semiconductor Research Corporation (SRC) under agreements through Contract No. 99-TJ-684, the National Science Foundation (NSF) under Grant Nos. CCR-9505472 and CCR-9803774, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of SRC, NSF, DARPA, or the United States Government.

**Keywords:** formal verification, model checking, abstraction, counterexample, refinement, binary decision diagram

## Abstract

As technology advances and demand for higher performance increases hardware designs are becoming more and more sophisticated. A typical chip design may contain over ten million switching devices. Since the systems become more and more complex, detecting design errors for systems of such scale becomes extremely difficult. Formal verification methodologies can potentially catch subtle design errors. However, many state-of-the-art formal verification tools suffer from the *state explosion problem*.

This thesis explores abstraction techniques to avoid the state explosion problem. In our methodology, *atomic formulas* extracted from an SMV-like concurrent program are used to construct *abstraction functions*. The initial abstract structure is built by using *existential abstraction* techniques. When the model checker disproves a universal property on the abstract structure, it generates a counterexample. However, this abstract counterexample might be spurious because abstraction is not complete. We provide a new symbolic algorithm to determine whether an abstract counterexample is spurious. When a counterexample is identified to be spurious, the algorithm will compute the shortest prefix of the abstract counterexample that does not correspond to an actual trace in the concrete model. The last abstract state in this prefix is split into less abstract states so that the spurious counterexample is eliminated. Thus, a more refined abstraction function is obtained. It is usually desirable to obtain the coarsest refinement which eliminates the counterexample because this corresponds to the *smallest* abstract model that avoids the spurious counterexample. We prove, however, that finding the coarsest refinement is NP-hard. Because of this, we use a polynomial-time algorithm which gives a suboptimal but sufficiently good refinement of the abstraction function. The applicability of our heuristic algorithm is confirmed by our experiments. Using the refined abstraction function obtained in this manner, a new abstract model is built and the entire process is repeated. Our methodology is complete for $\mathbf{ACTL}$, i.e., we are guaranteed to either find a valid counterexample or prove that the system satisfies the desired property.

On the other hand, this thesis also discusses a new data structure - abstract BDDs. Intuitively, an abstract BDD is obtained from a BDD by collapsing paths that have the same abstract value with respect to some abstraction function. There are many ways to collapse the paths corresponding to different types of abstract BDDs. We identify four types of abstract BDDs : S-type, 0-type, 1-type and $\vee$-type abstract BDDs for different applications. In this thesis, we show three applications of abstract BDDs. First, we will show how to check inequivalence between two combinational circuits using S-type and 0-type abstract BDDs. Then, we describe a methodology to generate an initial variable ordering using 0-type abstract BDDs. Finally, we demonstrate how to represent abstract Kripke structures using $\vee$-type abstract BDDs. Our experiments clearly show the efficiency of abstract BDDs. We believe that abstract BDDs can be applied to many other applications as well.

# Acknowledgements

When I came to CMU's open house of the ECE department four years ago, I had little idea about formal methods or model checking. The only thing I knew was that I was interested in logic and verification problems. Ed Clarke, my future advisor, said to me, "You can learn it!" Since then he has been painstakingly teaching me about model checking and formal methods from scratch. Whenever I lost track, Ed was there for help. I am always amazed by his enthusiasm and insistence on solving hard problems. In a word, without Ed's direction and encouragement, I would have buried myself in the maze of graduate study.

I owe a lot of gratitude to my thesis committee for their advice and patient reading of my dissertation. Masahiro Fujita has been long-time tutor and guide for my research. His expertise and advice kept me away from pitfalls. Every time I talked with Randy Bryant, I found interesting topics to work on. He has been a model for successful research through my whole graduate career. Don Thomas provided me with many insightful pieces of advice on my thesis; meeting with him is always an enjoyable experience.

This work would not be possible without the environment of our model checking group. Ed creates an open and cooperative research environment for students. It is very easy for me to ask around and get help. I have benefited from most members of the group: Xudong, Sergio, Will, Vicky, Marius, Sergey, Pankaj, Chaki, Anubhav and Alex. I also benefit a lot from visitors to our group, including Orna, Marco, Yunshan, Armin, Wolfgang, Poul and Deharbe. I not only appreciate their help, but also enjoy being with them.

Among my CMU colleagues, I spent most time working with Somesh Jha. He never runs out of research ideas. It was him who motivated me to work on abstract BDDs. My most fruitful year was spent with Helmut Veith and Dong Wang when Helmut visited Ed's group as a visiting professor. I will always remember our routinely eight o'clock night meeting at the CS student lounge. Helmut's knowledge and personality really inspire me to be a better researcher.

I spent two summers in Fujitsu Lab of America. The experience was incredible. As my supervisor, Jawahar Jain and Sree Rajan not only taught me how to do research but also how to enjoy doing research. I had a great time working and talking with Raj, Rajiv, Vamsi, and Bob. I would also like to thank Steve German from IBM and Richard Raimi from BIPS for discussions on many problems.

Though I am an ECE student, my office is in CS department. I have gotten help from staff in both departments including Catherine Copetas, Elaine Lawrence, Roxann Martin, Lynn Philibin, etc. It is you who make CMU so special.

I am very grateful to my parents and my sister. Their lifelong support and encouragement are the source of strength I have. I thank John He, Tim Zhou, Qiu Jian, Yi Wei, and Peter Fang for accompanying me through difficult times. Finally, the biggest thanks to Keyuan for her patience, understanding and the-best-of-world cooking. You are the best I have ever had.

# Contents


**1 Introduction** **4**

1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.2 Scope of the thesis . . . . . . . . . . . . . . . . . . . . . . . 7

1.3 Related research . . . . . . . . . . . . . . . . . . . . . . . . . 10

1.3.1 Abstraction for Model Checking . . . . . . . . . . . . 10

1.3.2 Counterexample-guided refinement . . . . . . . . . . 11

1.3.3 Other abstraction-refinement techniques . . . . . . . . 12

1.3.4 BDDs and abstraction . . . . . . . . . . . . . . . . . 13

1.4 Outline of this dissertation . . . . . . . . . . . . . . . . . . . 15

**2 Existential abstraction for $\mathrm{ACTL}^{\star}$** **16**

2.1 Kripke structures and $\mathrm{CTL}^{\star}$ . . . . . . . . . . . . . . . . . . . 17

2.2 Overview of BDDs . . . . . . . . . . . . . . . . . . . . . . . . 22

2.3 Concurrent Programs . . . . . . . . . . . . . . . . . . . . . . 25

2.4 Existential abstraction . . . . . . . . . . . . . . . . . . . . . . 27

2.5 Approximation for existential abstraction . . . . . . . . . . . 32

2.6 Remaining Problems . . . . . . . . . . . . . . . . . . . . . . . 35

**3 Counterexample-guided Abstraction Refinement** **36**

3.1 Generating the initial abstraction . . . . . . . . . . . . . . . . 37

3.2 Model checking the abstract model . . . . . . . . . . . . . . . 42

3.2.1 Identification of spurious path counterexamples . . . . 42
3.2.2 Identification of spurious loop counterexamples . . . . 46
3.3 Refining the abstraction . . . . . . . . . . . . . . . . . . . . . . 54
3.4 Performance improvements . . . . . . . . . . . . . . . . . . . . 59
3.4.1 Detecting more real counterexamples . . . . . . . . . 60
3.4.2 Abstraction for approximation . . . . . . . . . . . . . 61
3.4.3 Abstractions for distant variables . . . . . . . . . . . . 62
3.5 Experimental results . . . . . . . . . . . . . . . . . . . . . . . 63
3.5.1 Experiments on benchmark circuits . . . . . . . . . . 63
3.5.2 Debugging a multimedia processor . . . . . . . . . . 65
3.6 Proofs for Refinement Theorem . . . . . . . . . . . . . . . . . 70

**4 Refinement for General ACTL Counterexamples 77**
4.1 What are Counterexamples? . . . . . . . . . . . . . . . . . . 78
4.2 Tree-like Kripke structures . . . . . . . . . . . . . . . . . . . 80
4.2.1 Indexed Kripke structures . . . . . . . . . . . . . . . . 84
4.3 Generating tree-like counterexamples for ACTL . . . . . . . 85
4.3.1 Fixpoint Characterization for ACTL . . . . . . . . . 87
4.3.2 Algorithms to generate tree-like counterexamples . . . 90
4.4 Refinement algorithm for ACTL . . . . . . . . . . . . . . . . 96

**5 Abstract BDDs 100**
5.1 Abstract Binary Decision Trees . . . . . . . . . . . . . . . . . 100
5.2 S-type Abstract BDDs . . . . . . . . . . . . . . . . . . . . . . 102
5.3 0-type And 1-type Abstract BDDs . . . . . . . . . . . . . . . 109
5.4 $\vee$-type Abstract BDDs . . . . . . . . . . . . . . . . . . . . . . 113
5.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116

**6 Applications of abstract BDDs** **117**

6.1 Equivalence checking using abstract BDDs . . . . . . . . . . 117

6.2 Improving variable ordering using 0-type abstract BDDs . . . 121

6.3 Model Checking Using $\vee$-type Abstract BDDs . . . . . . . . . 129

6.3.1 Abstraction for $\mathbf{ACTL}^\star$ . . . . . . . . . . . . . . . . . 129

6.3.2 Case studies . . . . . . . . . . . . . . . . . . . . . . . . 131

6.3.3 Abstraction for Variable Ordering . . . . . . . . . . . 138

**7 Conclusion and Future Work** **141**

7.1 Using other data structures instead of BDDs . . . . . . . . . . 142

7.2 Implementing abstraction inside BDDs . . . . . . . . . . . . . 143

# Chapter 1

# Introduction

## 1.1 Background

Hardware designs are becoming more sophisticated as technology advances and demand for higher performance increases. A typical chip design may contain over ten million switching devices. As the systems become more and more complex, detecting design errors for systems of such scale becomes extremely difficult. It is common that even experienced engineers overlook some "corner" cases in the design phase. Ignoring such cases may result in serious problems. Currently, most designers still heavily depends on random simulation techniques to look for subtle bugs. However, it is known that simulation techniques often fail to reveal such subtle errors during the debugging phase.

In contrast to simulation, formal verification techniques have the capability to find subtle bugs. A number of researchers have proposed formal techniques including theorem proving [15, 92, 95], and model checking [28, 90, 91], etc. In theorem proving, the designer constructs a mathematical proof, with the aid of some automated support, that a model or a structure meet their specification. It is possible for these techniques to model systems at almost any level of detail. In particular, one can model systems with infinite state space and prove properties of entire classes of systems. The main drawback of theorem proving is that it requires great effort and creativity on the part of the user. On the other hand, model checking methods restrict the model to be finite-state and use state space searching algorithms to check *automatically* that the specification is satisfied. Therefore, these approaches require less expertise to use.

In model checking, specifications are written in certain *temporal logics* [42]. Pnueli [90] was the first to use temporal logic for reasoning about concurrent programs. Later, Clarke and Emerson [28] introduced computational tree logic ($\mathbf{CTL}$) and developed an efficient model checking algorithm. Their model checking algorithm includes three components: a formal model which describes the system to be verified, a specification of the correctness properties of the system, and a decision procedure to check whether the model satisfies the given specification. The model size is the major factor that affects the performance of decision procedures. This problem is commonly called the *state explosion problem*. Models with up to a million states can be verified using an explicit state model checking algorithm [29]. About a decade ago, McMillan proposed a symbolic model checking [78] algorithm using binary decision diagrams (BDDs) [17] (see Chapter 2.2 for a definition of BDDs). Burch, Clarke and McMillan discussed a number of improvements of the symbolic model checking algorithm [22]. By combining the new CTL model checking algorithm with the symbolic representation of state transition graphs, systems with extremely large number of states can be verified. The model checking system that McMillan developed as part of his Ph.D. thesis is called SMV [78]. It is based on a language for describing hierarchical finite-state concurrent systems. Programs in the language can be annotated by specifications expressed in temporal logic. The model checker extracts a transition system represented by BDDs from the SMV program and uses a BDD-based symbolic model checking algorithm to determine whether the system satisfies its specification (see

Figure 1.1). If the transition system, formally modeled as *Kripke structure* [57], does not satisfy some specification, the model checker will produce an execution trace that shows why the specification is false. This execution trace is called a *counterexample* for the specification. Symbolic model checking algorithms can typically verify designs with a few hundred symbolic variables.

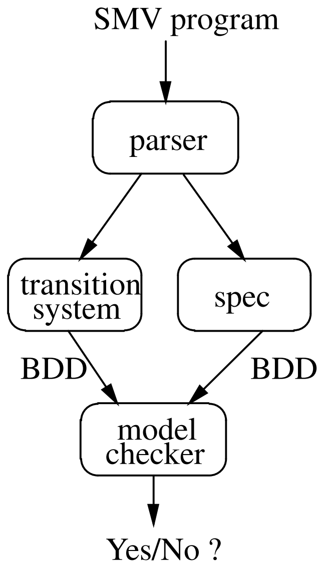



Figure 1.1: Model checker for SMV programs

More recently, propositional satisfiability [6, 49, 83, 98] based symbolic model checking algorithms [10] have been investigated for even larger designs. However, state-of-the-art hardware designs include hundreds of thousands of variables and the number of states in models grows exponentially in the number of variables. Therefore, applying model checking to large industrial designs is still a hard problem.

On the other hand, a number of state reduction approaches have been proposed to reduce the number of states under verification. State reduction techniques include symmetry reductions [43, 44, 58, 62], partial order reductions [51], and abstraction techniques [34, 31, 75]. Among these techniques, abstraction is the most general technique for handling the state explosion problem. In fact, it is essential for verifying designs of industrial complexity. Currently, abstraction is typically a manual process, often requiring considerable creativity and understanding of the problem domain. In order for model checking to be used more widely in industry, automatic techniques are needed for generating abstractions.

Intuitively, abstraction tries to simplify the models by hiding "irrelevant" details. Verifying the simplified models is in general more efficient than verifying properties of the original ones. Abstraction techniques can be classified as *over-approximation* [31, 68] or *under-approximation* techniques [70, 87]. Over-approximation techniques systematically release constraints, and thus add more behaviors to the system. They establish a relationship between the abstract model and the original one such that correctness at the abstract level implies correctness of the original system. In contrast, under-approximation techniques systematically remove irrelevant behaviors from the system. An under-approximation technique establishes a relationship between the abstract model and the original one, so that falseness at the abstract level will imply falseness of the original system.

## 1.2 Scope of the thesis

This thesis explores abstraction techniques to avoid the state explosion problem. The techniques follow the general framework established by Clarke, Grumberg, and Long [31] which is known as *existential abstraction*. Existential abstraction is an over-approximation technique. Given a concrete Kripke structure, an *abstract* Kripke structure is built according to a given abstraction function. If a property holds on the abstract structure, then it also holds on the concrete structure. However, it is usually computationally hard to construct abstract structures. Therefore, we often need to approximate an abstract structure instead of directly building it. Clarke, Grumberg and Long defined a fast and simple approximation technique which approximates the abstract structure efficiently. However, there exist several unsolved problems in their approach:

1. It is not known how to generate abstraction functions automatically.

2. The abstraction is conservative but not complete. When a property is false on the abstract structure, it may still be valid for the concrete structure.

3. Approximation introduces many spurious transitions, i.e., abstract transitions which do not correspond to concrete transitions. It is unknown how to reduce the number of spurious transitions.

4. In some cases, it is hard to build the BDDs for the abstraction functions. In these cases, constructing abstract structures becomes extremely hard.

The goal of this thesis is to attack these problems. We have proposed a counterexample-guided abstraction refinement methodology which addresses the problems (1) and (2). We have also introduced a new data structure – *abstract BDDs* to alleviate the problems (3) and (4). The principle contributions of this thesis are detailed below:

**A counterexample-guided automatic abstraction-refinement method:** In this methodology, *atomic formulas* extracted from an SMV program are used to construct *abstraction functions*. The initial abstract model is built by using the existential abstraction techniques. When the model checker disproves an $\mathbf{ACTL}^\star$ property on the abstract structure, it generates a counterexample. However, this abstract counterexample might not be valid because abstraction is not complete. We say that such a counterexample is *spurious*. We provide a new symbolic algorithm to determine whether an abstract counterexample is spurious. When a counterexample is identified to be spurious, the algorithm will compute the shortest prefix of the abstract counterexample that does not correspond to an actual trace in the concrete model. The last abstract state in this prefix is split into less abstract states so that the spurious counterexample is eliminated. Thus, a more refined abstraction function is obtained. Note that there may be many ways of splitting the abstract state; each determines a different refinement of the abstraction function. It is desirable to obtain the coarsest refinement which eliminates the counterexample because this corresponds to the *smallest* abstract model that avoids the spurious counterexample. We prove, however, that finding the coarsest refinement is NP-hard. Because of this, we use a polynomial-time algorithm which gives a suboptimal but sufficiently good refinement of the abstraction function. The applicability of our heuristic algorithm is confirmed by our experiments. Using the refined abstraction function obtained in this manner, a new abstract model is built and the entire process is repeated. Our methodology is complete for $\mathbf{ACTL}$, i.e., we are guaranteed to either find a valid counterexample or prove that the system satisfies the desired property. In principle, our methodology can be extended to all of $\mathbf{ACTL}^\star$.

**Abstract BDDs based verification technologies:** Intuitively, an abstract BDD is obtained from a BDD by collapsing paths that have the same abstract value with respect to some abstraction function. There are many ways to collapse the paths corresponding to different types of abstract BDDs. We identify four types of abstract BDDs : S-type, 0-type, 1-type and $\vee$-type abstract BDDs for different applications. In this thesis, we show three applications of abstract BDDs. First, we will show how to check inequivalence between two combinational circuits using S-type and 0-type abstract BDDs. Then, we describe a methodology to generate an initial variable ordering using 0-type abstract

BDDs. Finally, we demonstrate how to represent abstract Kripke structures using $\vee$-type abstract BDDs. Our experiments clearly show the efficiency of abstract BDDs. We believe that abstract BDDs can be applied to many other applications as well.

## 1.3 Related research

### 1.3.1 Abstraction for Model Checking

Many abstraction techniques can be viewed as application of abstract interpretation [34, 99]. Given an abstract domain, abstract interpretation provides a general framework for automatically "interpreting" systems on an abstract domain. The classical abstract interpretation framework is used to prove safety properties, and does not consider temporal logic or model checking. For example, Bjorner, Browne and Manna use abstract interpretation to automatically generate invariants for general infinite-state systems [12]. Later, in [31], the authors have proposed an abstract interpretation methodology for $\mathbf{ACTL}^{\star}$ properties. Abstraction techniques for various fragments of $\mathbf{CTL}^{\star}$ have been discussed in [36, 37]. These abstraction techniques have been extended to the $\mu$-calculus [35, 74].

Abstraction techniques for infinite state systems are crucial for successful verification [2, 7, 71, 77]. Graf and Saïdi [54] have proposed *predicate abstraction* techniques to abstract an infinite state system into a finite state system. Later, a number of optimization techniques have been developed in [8, 38]. Saïdi and Shankar have integrated predicate abstraction into the PVS system which could easily determine when to abstract and when to model check [97]. Colón and Uribe [33] have presented a way to generate finite-state abstractions using a decision procedure. Similar to predicate abstraction, their abstraction is generated using abstract boolean variables. One difference between our approach and the predicate abstraction oriented approaches is that the latter tries to build an abstract model on-the-fly while traversing the reachable state sets. Our approach tries to build the abstract transition relation directly.

Wolper and Lovinfosse [105] have verified *data independent* systems using model checking. In a data independent system, the data values never affect the control flow of the computation. Therefore, the datapath can be abstracted away entirely. Van Aelten et al [3] have discussed a method for simplifying the verification of synchronous processors by abstracting away the data path. Abstracting the datapath using uninterpreted function symbols is very useful for verifying pipeline systems [9, 21, 20, 21, 64, 103, 104]. A number of researchers have modeled or verified industrial hardware systems using abstraction techniques [47, 53, 55, 56]. In many cases, their abstractions are generated manually and combined with theorem proving techniques [95, 96]. Dingel and Filkorn have used data abstraction and assume-guarantee reasoning combined with theorem proving techniques to verify infinite state systems [39]. Recently, McMillan has incorporated a new type of data abstraction, assume-guarantee reasoning and theorem proving techniques in his Cadence SMV system [79].

### 1.3.2 Counterexample-guided refinement

Using counterexamples to refine abstract models has been investigated by a number of other researchers beginning with the *localization reduction* of Kurshan [67, 68]. He models a concurrent system as a composition of $L$-processes $L_1, \ldots, L_n$ ($L$-processes are described in detail in [68]). The localization reduction is an iterative technique that starts with a small subset of relevant $L$-processes that are topologically close to the specification in the *variable dependency graph.* All other program variables are released by nondeterministic assignments. If the counterexample is found to be spurious, additional variables are added to eliminate the counterexample. The heuristic for selecting these variables also uses information from the variable dependency graph. Note that the localization reduction either leaves a variable unchanged or replaces it by a nondeterministic assignment. A similar approach has been described by Balarin in [5].

In this thesis, we propose a new counterexample-guided refinement technique using existential abstraction. In our approach, the abstraction functions exploit logical relationships among variables appearing in atomic formulas that occur in the control structure of the program. Moreover, the way we use abstraction functions makes it possible to distinguish many degrees of abstraction for each variable. Therefore, in the refinement step only very small and local changes to the abstraction functions are necessary and the abstract model remains comparatively small. Recently, Lakhnech and his colleagues have also used counterexamples to refine abstraction for infinite systems [69].

### 1.3.3 Other abstraction-refinement techniques

Another refinement technique has recently been proposed by Lind-Nielson and Andersen [72, 73]. Their model checker uses upper and lower approximations in order to handle all of CTL. Their approximation techniques enable them to avoid rechecking the entire model after each refinement step while guaranteeing completeness. As in [5, 68], the variable dependency graph is used to obtain the initial abstraction as well as in the refinement process. Variable abstraction is also performed in a similar manner. Therefore, our abstraction-refinement methodology relates to their technique in essentially the same way as it relates to the classical localization reduction.

A number of other papers [70, 86, 87] have proposed abstraction-refinement techniques for CTL model checking. These techniques use the

BDD size as abstraction criterion. When the BDD size exceed a certain limit, the abstraction is applied. Govindaraju and Dill [52] tries to identify the first spurious state in an abstract counterexample. It randomly chooses a concrete state corresponding to the first spurious state and tries to construct a real counterexample starting with the image of this state under the transition relation. The paper only talks about safety properties and path counterexamples. It does not describe how to check liveness properties with cyclic counterexamples. Furthermore, our method does not use random choice to extend the counterexample; instead it analyzes the cause of the spurious counterexample and uses this information to guide the refinement process.

### 1.3.4 BDDs and abstraction

As a data structure for symbolic representation, BDDs [17] have been widely used in synthesis, verification, validation, [16, 65, 101]. BDDs are directed acyclic graphs (DAGs) with two terminal nodes labeled by 0 and 1 respectively. For fixed variable orders, BDDs are a canonical representation for Boolean functions. BDDs are often substantially more compact than conjunctive or disjunctive normal forms. Different variable orders result in BDDs with different sizes [18]. Finding good variable ordering is the central problem for applying BDDs effectively [18, 13]. Numerous heuristics have been proposed to address this problem. *Topology based* or *static* variable ordering techniques (for example, using depth-first or breadth-first search of the circuits) have been extensively investigated for more than a decade [45, 76, 4]. However, these techniques often perform poorly because they rely on purely structural information of the circuits. *Sifting-based* dynamic ordering techniques are more popular [46, 85, 94] because they can dynamically change the variable orders during the course of the computation. However, they are extremely expensive with respect to both time and space. Moreover, during the reordering of the BDDs, these techniques can frequently get stuck in a local minimum and thus fail to reduce the size of the resulting graph to an acceptable degree. Other optimization techniques such as simulated annealing have also been applied to dynamic reordering as well [14, 41]. However, these approaches are usually even slower than sifting-based techniques.

Sampling based approaches have been proposed by Meinel and Slobodov [81] and Jain, et al. [60] to overcome these problems. In Jain's approach, a portion of the Boolean space for the output function is analyzed using reordering techniques. This sampled subspace is obtained by restricting the Boolean function using cubes. Cubes are monomials of a subset of the variables. This order is then used for analyzing the complete Boolean space of the given function. By appropriately using the (limited) global information about the given function, the local minimum problem of current sifting-based ordering techniques is reduced. However, using only *randomly generated subspaces* for samplings has several practical problems [80, 82]. First, this cube based sampling technique tends to generate less efficient variable orders. Secondly, the generated variable orders can vary dramatically between different runs. This causes an extremely large variance in the quality of the results and makes cube based sampling difficult to automate effectively.

In order to verify specific types of hardware designs, for example, arithmetic circuits, variation of BDDs have been proposed. Clarke et al. [32] proposed MTBDDs which are similar to ordinary BDDs except that the terminal nodes can be arbitrary integer values instead of 0 and 1. Bryant and Chen developed BMDs [19] which give a compact representation for certain functions that have MTBDDs of exponential size. Using Kronecker products, Clarke, Fujita and Zhao proposed Hybrid Decision Diagrams (HDDs) [30] which can be more concise than both MTBDDs and BMDs. Other variations of BDDs are extensively investigated for different applications as well [40, 50, 63, 61].

In [31], Clarke, Grumberg and Long used the Chinese Remainder Theorem to prove properties of arithmetic circuits. They choose residue functions ($h(x) = x \mod p$) as abstraction functions. Later, Kimura extended the idea and proposed Residue BDDs [66] for verifying combinational multipliers. More recently, residue ADDs [93] have been proposed to verify combinational multipliers as well. In this thesis, we generalize the idea of residue BDDs by allowing arbitrary abstraction functions, obtaining *abstract BDDs* (aBDDs). Therefore, residue BDDs are a special case of abstract BDDs.

## 1.4 Outline of this dissertation

This thesis is organized as follows: In Chapter 2, we introduce basic definitions for concurrent programs, the theory of Kripke structures, temporal logic $\mathbf{CTL}^\star$ and the theory of existential abstraction. In Chapter 3, we describe a counterexample-guided abstraction refinement framework for a fragment of $\mathbf{ACTL}^\star$. In Chapter 4, we discuss how to extend this methodology to refine the abstraction for other $\mathbf{ACTL}^\star$ properties. Abstract BDDs are defined in Chapter 5. Applications of different abstract BDDs are discussed in Chapter 6.

# Chapter 2

# Existential abstraction for ACTL$^\star$

Because of the state explosion problem, successful verification usually requires state reduction techniques. Abstraction techniques are one of the most general state reduction techniques. Intuitively, abstraction techniques remove "irrelevant" information from the design. Since the properties which need to be verified usually depend only on a part of the design, abstraction techniques can be very useful to reduce the state space. For example, for a data independent circuit [105], the data paths can usually be ignored. There are many different ways to obtain an abstract state space. They can be classified into under-approximation approaches [70, 87] and over-approximation approaches [31, 68]. Existential abstraction is an over-approximation approach that can also be viewed as an application of abstract interpretation [34]. In existential abstraction, a function $h$ maps each state in the state space to an abstract state in a typically smaller abstract state space. Two states are equivalent with respect to the abstraction function $h$ if they are mapped to the same abstract state. This equivalence relation partitions the state space into equivalence sets. Thus, abstract states are essentially equivalence classes of states. For example, consider the simple traffic light controller in Figure 2.1(a). Assume that we have an abstraction function $h(\texttt{red}) = \texttt{red}$, $h(\texttt{green}) = \texttt{go}$ and $h(\texttt{yellow}) = \texttt{go}$. Then the obtained abstract structure is shown in Fig-

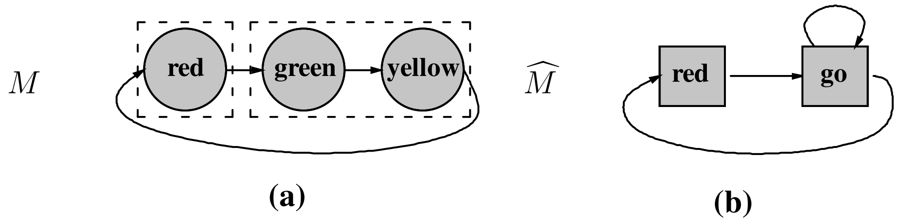



Figure 2.1: Abstraction of a Traffic Light.

ure 2.1(b).

In Chapter 2.1, we will first describe Kripke structures, and computational tree logic ($\mathbf{CTL}^\star$). Kripke structures are used to model finite automata while $\mathbf{CTL}^\star$ are used to specify properties on the finite automata. $\mathbf{ACTL}^\star$ is a fragment of $\mathbf{CTL}^\star$ which allows only universal path quantification. In Chapter 2.2, we provide an overview of BDDs. In Chapter 2.3, we provide several notations related to concurrent programs. In Chapter 2.4 and Chapter 2.5, we will discuss basic existential abstraction theory. In Chapter 2.6, we will describe limitation of the state-of-the-art existential abstraction technique which essentially motivates this thesis.

## 2.1 Kripke structures and $\mathbf{CTL}^\star$

Given a set of atomic formulas $A_t$, a *Kripke structure* $M$ is a 4-tuple, i.e., $M = (S, I, R, L)$, where $S = D$ is the set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation and $L : S \to 2^{A_t}$ is the labeling function.

Given a Kripke structure with a single initial state, the corresponding *computational tree* is obtained by unwinding the structure into an infinite tree with the initial state at the root. Intuitively, computational tree logic ($\mathbf{CTL}^\star$) is composed of formulas which describe properties of the computational trees. $\mathbf{CTL}^\star$ formulas include *path quantifiers* and *temporal operators*. There are two kinds of path quantifiers: $\mathbf{A}$ ("for all computation paths") and $\mathbf{E}$ ("for some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths staring at that state have some property. The temporal operators describe properties of a path through the tree. There are five basic operators:

- $\mathbf{X}\ \varphi$ ("next time") requires that property $\varphi$ holds in the second state of the path.
- $\mathbf{F}\ \varphi$ ("eventually" or "in the future") asserts that property $\varphi$ will hold at some state on the path.
- $\mathbf{G}\ \varphi$ ("always" or "globally") specifies that property $\varphi$ holds at every state on the path.
- $\mathbf{U}$ ("until") is used to combine two properties. $\varphi\,\mathbf{U}\ \psi$ holds if there is a state on the path where property $\psi$ holds, and at every preceding state on the path, property $\varphi$ holds [26].

**Syntax** There are two types of formulas in $\mathbf{CTL}^{\star}$: *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). The syntax of both state and path formulas is given by the following rules:

- If $p \in A_t$, then $p$ is a state formula.
- If $f$ and $g$ are state formulas, then $\neg f$, $f \wedge g$ and $f \vee g$ are state formulas.
- If $f$ is a path formula, then $\mathbf{E}\ f$ and $\mathbf{A}\ f$ are state formulas.
- If $f$ is a state formula, then $f$ is also a path formula.
- If $f$ and $g$ are state formulas, then $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\,\mathbf{U}\ g$ and $f\,\mathbf{R}\ g$ are path formulas.

**Semantics** A path $\pi$ in $M$ is an infinite sequence of states, $\pi = \langle s_0, s_1, \ldots \rangle$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. We use $\pi^i$ to denote the suffix of $\pi$ starting at $s_i$. If $f$ is a state formula, the notation $M, s \models f$ means that $f$ holds at state $s$ in the Kripke structure $M$. Similarly, if $f$ is a path formula, $M, \pi \models f$ means that $f$ holds along path $\pi$ in $M$. The relation $\models$ is defined recursively as follows (assuming that $f_1$ and $f_2$ denote state formulas and $g_1$ and $g_2$ denote path formulas):

$$
\begin{array}{lcl}
M, s \models p & \Leftrightarrow & p \in L(s); \\
M, s \models \neg f_1 & \Leftrightarrow & M, s \not\models f_1; \\
M, s \models f_1 \wedge f_2 & \Leftrightarrow & M, s \models f_1 \text{ and } M, s \models f_2; \\
M, s \models f_1 \vee f_2 & \Leftrightarrow & M, s \models f_1 \text{ or } M, s \models f_2; \\
M, s \models \mathbf{E}\ g_1 & \Leftrightarrow & \text{there exists } \pi = \langle s, \ldots \rangle \text{ such that } M, \pi \models g_1; \\
M, s \models \mathbf{A}\ g_1 & \Leftrightarrow & \text{for every } \pi = \langle s, \ldots \rangle \text{ it holds that } M, \pi \models g_1; \\
M, \pi \models f_1 & \Leftrightarrow & \pi = \langle s, \ldots \rangle \text{ and } M, s \models f_1; \\
M, \pi \models \neg g_1 & \Leftrightarrow & M, \pi \not\models g_1; \\
M, \pi \models g_1 \wedge g_2 & \Leftrightarrow & M, \pi \models g_1 \text{ and } M, \pi \models g_2; \\
M, \pi \models g_1 \vee g_2 & \Leftrightarrow & M, \pi \models g_1 \text{ or } M, \pi \models g_2; \\
M, \pi \models \mathbf{X}\ g_1 & \Leftrightarrow & M, \pi^1 \models g_1; \\
M, \pi \models \mathbf{F}\ g_1 & \Leftrightarrow & \exists k \geq 0, \text{such that } M, \pi^k \models g_1; \\
M, \pi \models \mathbf{G}\ g_1 & \Leftrightarrow & \forall k \geq 0, \text{it holds that } M, \pi^k \models g_1; \\
M, \pi \models g_1 \,\mathbf{U}\ g_2 & \Leftrightarrow & \exists k \geq 0, \text{such that } M, \pi^k \models g_2 \text{ and } \forall 0 \leq j < k, \text{it holds} \\
 & & \text{that } M, \pi^j \models g_1; \\
M, \pi \models g_1 \,\mathbf{R}\ g_2 & \Leftrightarrow & \forall j \geq 0, \text{if } \forall i < j, M, \pi^i \not\models g_1, \text{then } M, \pi^j \models g_2.
\end{array}
$$

We assume that the specifications are written in a fragment of $\mathbf{CTL}^\star$ called $\mathbf{ACTL}^\star$ (see [31]), which eliminates the ability to describe the existence of a path, i.e., the $\mathbf{E}$ path quantifier. $\mathbf{ACTL}^\star$ is the fragment of $\mathbf{CTL}^\star$ where negation is restricted to the atomic level, and path quantification is restricted to universal path quantification. Formally, it is defined as follows.

**Definition 2.1.1** The logic $\mathbf{ACTL}^\star$ over a program $P$ is the fragment of $\mathbf{CTL}^\star$ which contains the set of formulas given by the following inductive definition:

- For each variable $x_i$ and element $d$ of $D_{x_i}$, $x_i = d$ is an atomic formula.

- If $\varphi$ is an atomic formula, then $\neg\varphi$ is a formula.
- If $\varphi$ and $\psi$ are formulas, then $\varphi \wedge \psi$ and $\varphi \vee \psi$ are formulas.
- All the state formulas are also path formulas.
- If $\varphi$ is a path formula, then $\mathbf{A}\ \varphi$ is a state formula.
- If $\varphi$ and $\psi$ are path formulas, then $\mathbf{X}\ \varphi$, $\mathbf{F}\ \varphi$, $\varphi\,\mathbf{U}\ \psi$, and $\varphi\,\mathbf{R}\ \psi$ are also path formulas.

For example, $\mathbf{AG}(\mathtt{req} = 1 \Rightarrow\ \mathbf{AF}\ \ \mathtt{ack} = 1)$ is an $\mathbf{ACTL}^{\star}$ formula while $\mathbf{AG}\,\mathbf{EF}\,\mathtt{comp} = 1$ is not an $\mathbf{ACTL}^{\star}$ formula. Note that in $\mathbf{ACTL}^{\star}$, negation ($\neg$) is only applied to atomic formulas. For a given $\mathbf{ACTL}^{\star}$ specification $\varphi$, we define $\mathrm{Atoms}(\varphi)$ to be the set of atomic formulas appearing in the specification $\varphi$. For example, $\mathrm{Atoms}(\mathbf{AG}(\mathtt{req} = 1 \Rightarrow\ \mathbf{AF}\ \ \mathtt{ack} = 1)) = \{\mathtt{req} = 1, \mathtt{ack} = 1\}$.

In order to alleviate the state explosion problem, it is desirable to develop techniques that replace a large structure by a smaller structure that satisfies the same properties. In the following, we will consider two kinds of relations between Kripke structures: *bisimulation* equivalence and *simulation* preorders.

Given two Kripke structures, $M = (S, I, L, R)$ and $M' = (S', I', L', R')$ over the same set of atomic formulas $A_t$, a relation $B \subseteq S \times S'$ is a *bisimulation relation* between $M$ and $M'$ if and only if for all $(s, s') \in B$, the following conditions hold:

- $L(s) = L'(s')$.
- For each state $s_1$ such that $(s, s_1) \in R$, there exists $s'_1$ such that $(s', s'_1) \in R'$ and $(s_1, s'_1) \in B$.

- For each state $s'_1$ such that $(s', s'_1) \in R'$, there exists $s_1$ such that $(s, s_1) \in R$ and $(s_1, s'_1) \in B$.

The structure $M$ and $M'$ are *bisimulation equivalent* (denoted $M \equiv M'$) if there exists a bisimulation relation $B$ such that for every initial state $s_0 \in I$ there exists an initial state $s'_0 \in I'$ such that $(s_0, s'_0) \in B$. In addition, for every initial state $s'_0 \in I'$ there exists an initial state $s_0 \in I$ in $M$ such that $(s_0, s'_0) \in B$.

**Theorem 2.1.1** [26] *If $M \equiv M'$ then for every* $\mathbf{CTL}^\star$ *formula $\varphi$,*

$$M \models \varphi \text{ if and only if } M' \models \varphi.$$

Therefore, instead of checking a property $\varphi$ on $M$, we can check the property on the bisimilar structure $M'$. Sometimes it is hard to find a smaller bisimilar structure. In order to achieve greater reductions, the *simulation preorder* relation is introduced. Simulation is closely related to bisimulation. While bisimulation guarantees that two structures have the same behaviors, simulation relates a structure to an *abstraction* of the structure. Since the abstraction can hide some details of the orginal structure, it might generate smaller structures. Intuitively, simulation guarantees that every behavior of a structure is also a behavior of its abstraction. However, the abstraction might have behaviors that are not possible in the original structure.

Given two Kripke structures $M$ and $M'$ with $A_t \supseteq A'_t$, a relation $H \subseteq S \times S'$ is a *simulation relation* between $M$ and $M'$ if and only if for all $(s, s') \in H$, the following conditions hold.

- $L(s) \cap A'_t = L'(s')$.
- For each state $s_1$ such that $(s, s_1) \in R$, there exists a state $s'_1$ with the property that $(s', s'_1) \in R'$ and $(s, s'_1) \in H$.

We say that $M'$ *simulates* $M$ (denoted by $M \preceq M'$) if there exists a simulation relation $H$ such that for every initial state $s_0$ in $M$ there exists an initial state $s'_0$ in $M'$ for which $(s_0, s'_0) \in H$.

**Theorem 2.1.2** [26] *If $M \preceq M'$, then for every* $\mathbf{ACTL}^\star$ *formula $\varphi$ (with atomic propositions in $A'_t$), $M' \models \varphi \Rightarrow M \models \varphi$.*

**Theorem 2.1.3** [26] *$\preceq$ is a preorder on the set of structures.*

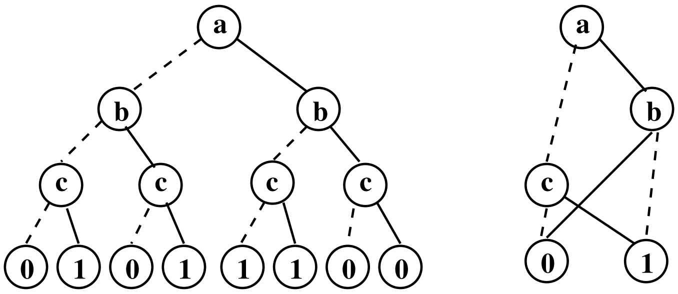
Similar results for $\omega$-automata are discussed in Kurshan's book [68].

## 2.2 Overview of BDDs

Given a boolean function $f : B^n \rightarrow B$ (Let $B = \{0, 1\}$ denote the Boolean domain), a **binary decision tree (BDT)** can be used to represent its truth table. A BDT consists of two types of nodes: internal nodes and terminal nodes. Every internal node is labeled by a variable and has two outgoing edges toward two lower level nodes (closer to the terminal nodes): the 0-edge corresponds to the case where the variable is assigned 0, and 1-edge corresponds to the case where the variable is assigned 1. Every terminal node is labeled "0" or "1". The *subgraph* of an internal node is the subtree corresponding to that node.

As an example, the truth table for $f(a, b, c) = (a \wedge \neg b) \vee (\neg a \wedge c)$ is shown in Figure 2.2(a). Figure 2.2(b) contains the corresponding BDT (dashed edges represent 0-edges and solid edges represent 1-edges). Notice that every variable appears exactly once in each path from the root to a terminal. If we fix the order of variables appearing along every path, the resulting graph is called an *ordered binary decision tree* (OBDT). It is easy to see that OBDTs are canonical; in other words, given an order of the variables, the OBDTs are unique representations of the given boolean functions.

| a | b | c | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |



**(a)** **(b)** **(c)**

Figure 2.2: BDT and BDD for $f = (a \wedge \neg b) \vee (\neg a \wedge c)$

An ordered binary decision diagram (OBDD) is a directed acyclic graph (DAG) representation obtained from OBDTs by applying the following three operations:

1. Merge all terminal 0 nodes into one terminal 0 node, and analogously for the terminal 1 nodes;

2. If two nodes are labeled by the same variable and their subgraphs are isomorphic, merge them into one node;

3. If a node A's two outgoing edges both point to the same node B, then delete node A and redirect the edges that had previously pointed to node A to node B.

As an example, the OBDD for the function in Figure 2.2(a) is shown in Figure 2.2(c). In this thesis, we follow the convention to write BDDs instead of OBDDs.

Given two BDDs $f$ and $g$, the resulting BDD $r$ of a boolean operation $f \circ g$ can be constructed using the *Shannon expansion*:

$$r = f \circ g = [\bar{x} \wedge (f_{x=0} \circ g_{x=0})] \vee [x \wedge (f_{x=1} \circ g_{x=1})]$$

where $x$ is the variable which comes first in the order among all the variables in $f$ and $g$, and $f_{x=0}$, $f_{x=1}$, $g_{x=0}$ and $g_{x=1}$ are the functions obtained when $x$ is restricted to the particular values, i.e. $0$ or $1$. They are also called *cofactor functions* of $f$ and $g$ respectively.

During the BDD construction, an *expansion* phase uses Shannon expansion to recursively divide the problem into smaller problems following the given variable order from the root to the terminals. After computing the cofactors, a *reduction* phase is called to reduce expressions to ensure uniqueness.

Bryant [17] showed that BDDs are also canonical and can be exponentially more compact than the corresponding truth tables or OBDTs. In addition, he showed that the BDD size is extremely sensitive to the variable ordering. The graph size of one ordering can be exponentially smaller than that of another ordering.

As an example, consider the boolean function $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$. When the variables are ordered by $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$, we only need eight nodes. However, when they are ordered by $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$, we need sixteen. Generalizing this function to a function over variables $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, it is easy to see that by using the first ordering we need $2n$ internal nodes while by using the second ordering we need $2(2^n - 1)$ internal nodes. This effect is dramatic for large values of $n$. Therefore, addressing this problem is important for BDD based applications though determining the optimal variable ordering is an NP-complete problem [13]. Various ordering heuristics have been proposed. In general, they can be classified into either *topology based* (static) or *functional analysis based* (dynamic) approaches.

Topology based variable ordering approaches (using for example depth-first or breadth-first search techniques) have been extensively investigated for more than a decade [45, 76]. Variables are usually grouped with respect to structural proximity. However, these techniques are not satisfactory. The problem is that the functionality of the designs is not taken into account.

In the dynamic variable ordering approach, one starts with an initial ordering, which is permuted during the BDD construction, such that the size of the resulting BDD is minimized [59, 94]. This approach is automatic and transparent to the users. In the *sifting* based dynamic ordering approach [94], periodic reordering of variables is attempted to reduce the memory requirements. Given a graph $G$, a variable $v$ is successively moved to each position in the ordering list and the resulting graph size is examined. The variable is finally assigned the position that results in the smallest graph size. Improvements to sifting based reordering techniques were proposed by [84], where the number of sift operations was reduced by sifting together the symmetric variable pairs. Further improvements were suggested in [85] where the concept of extended symmetry was introduced to group larger blocks of variables.

## 2.3 Concurrent Programs

In current technologies, hardware is usually described in hardware description languages, such as Verilog, VHDL and SMV etc. A *program* $P$ contains a finite set of variables $V = \{x_1, \cdots, x_n\}$, where each variable $x_i$ has an associated finite domain $D_{x_i}$. The set of all possible states for program $P$ is $D_{x_1} \times \cdots D_{x_n}$ which we denote by $D$. *Expressions* are built from variables in $V$, constants in $D_{x_i}$, and function symbols in the usual way, e.g. $x_1 + 3$. *Atomic formulas* are constructed from expressions and relation symbols, e.g. $x_1 + 3 < 5$. Similarly, *predicates* are composed of atomic formulas using negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). Given a predicate $p$, $\mathrm{Atoms}(p)$ is the set of atomic formulas occurring in it. Let $p$ be a predicate containing variables from $V$, and $d = (d_1, \ldots, d_n)$ be an element from $D$. Then we write $d \models p$ when the predicate obtained by replacing each occurrence of the variable $x_i$ in $p$ by the constant $d_i$ evaluates to true.

Let $A_t$ denote the set of atomic formulas in $P$. Then $P$ naturally corresponds to a *Kripke structure* $M = (S, I, R, L)$, where $S = D$ is the set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation and $L : S \to 2^{A_t}$ is the labeling function. Translating a program into a Kripke structure is straightforward and will not be described in this thesis. Details are available in [26].

Each variable $x_i$ in the program has an associated *transition block* $B_i$ [78], which defines both the initial value and the transition relation for the variable $x_i$. An example of a transition block for the variable $x_i$ is shown in Figure 2.3, where $I_i \subseteq D_{x_i}$ is the initial expression for the variable $x_i$, each condition $C_i^j$

| | |
|---|---|
| **init**($x_i$) := $I_i$; | **init**($x$) := $0$; |
| **next**($x_i$) := **case** | **next**($x$) := **case** |
| $C_i^1 : A_i^1$; | $reset = \mathrm{TRUE} : 0$; |
| $C_i^2 : A_i^2$; | $x < y : x + 1$; |
| $\cdots : \cdots$; | $x = y : 0$; |
| $C_i^k : A_i^k$; | **else** : $x$; |
| **esac**; | **esac**; |

Figure 2.3: A generic transition block and a typical example

is a predicate, and $A_i^j$ is an expression. The semantics of the transition block is similar to the semantics of the **case** statement in the modeling language of SMV, i.e., find the least $j$ such that in the current state condition $C_i^j$ is true and assign the value of the expression $A_i^j$ to the variable $x_i$ in the next state. For each $B_i$, we define $\mathrm{Atoms}(B_i)$ to be the set of atomic formulas that appear in the conditions, i.e., $\mathrm{Atoms}(B_i) = \bigcup \mathrm{Atoms}(C_i^j)$. Accordingly, we define $\mathrm{Atoms}(P)$ to be the set of atomic formulas appearing in $P$. Common hardware description languages like Verilog and VHDL can be compiled into this language.

An abstraction $h$ for a program $P$ is given by a surjection $h : D \rightarrow \widehat{D}$. Notice that the surjection $h$ induces an equivalence relation $\equiv_h$ on the domain $D$ in the following manner: let $d, e$ be states in $D$, then

$$d \equiv_h e \ \text{ iff } \ h(d) = h(e).$$

Since an abstraction can be represented either by a surjection $h$ or by an equivalence relation $\equiv_h$, we sometimes switch between these representations to avoid notational overhead.

**Auto-abstraction functions** The equivalence relation $\equiv_h$ partitions $D$ into a set of equivalence classes. This set is denoted by $[D]_{\equiv_h}$ and defined as $\{[d] \mid d \in D\}$ where $[d] = \{e \mid h(e) = h(d)\}$. For each $h$, we fix a function $rep_h : [D]_{\equiv_h} \rightarrow D$ that selects a *unique representative* from each equivalence class $[d]$. Thus, for a 0-1 vector $d \in D$, $rep_h([d])$ is the unique representative of the equivalence class $[d]$ of $d$.

The abstraction function $h$ induces a new abstraction function $\mathcal{H} : D \rightarrow D$ as follows:

$$\mathcal{H}(d) \ = \ rep_h([d]).$$

Since $\mathcal{H}$ operates on $D$, we call $\mathcal{H}$ the generated *auto-abstraction function*. From the definition of $\mathcal{H}$ it is easy to see that $\mathcal{H}(rep_h([d])) = rep_h([d])$ and $\mathcal{H}(\mathcal{H}(d)) = \mathcal{H}(d)$. Note that the image of $D$ under the function $\mathcal{H}$ is simply the set of representatives. This set of representatives will be denoted by $\mathit{Img}(\mathcal{H})$. Since $h$ was assumed to be a surjection, it follows that $|\mathit{Img}(\mathcal{H})| = |A|$.

## 2.4 Existential abstraction

Given a program $P$ and an $\mathbf{ACTL}^\star$ property $\varphi$, we obtain a Kripke structure $M = (S, I, R, L)$ where $S$, $I$ and $R$ are the same as traditional Kripke structure defined in Chapter 2.1. $L : S \to 2^{\mathrm{Atoms}(\varphi)}$ is a labeling function given by $L(d) = \{f \in \mathrm{Atoms}(\varphi) \mid d \models f\}$. Intuitively, the function $L$ labels each state by atomic formulas extracted from $\varphi$ because only these atomic formulas are relevant for verifying $\varphi$.

Assume that we are given a program $P$ and an abstraction function $h$ for $P$. The *abstract Kripke structure* $M_h = (S_h, I_h, R_h, L_h)$ corresponding to the abstraction function $h$ is defined as follows:

1. $S_h$ is the abstract domain $\widehat{D}$.
2. $I_h(\widehat{d})$ if and only if $\exists d(h(d) = \widehat{d} \wedge I(d))$.
3. $R_h(\widehat{d_1}, \widehat{d_2})$ if and only if $\exists d_1 \exists d_2(h(d_1) = \widehat{d_1} \wedge h(d_2) = \widehat{d_2} \wedge R(d_1, d_2))$.
4. $L_h(\widehat{d}) = \bigcup_{h(d)=\widehat{d}} L(d)$.

An atomic formula $f$ *respects* the abstraction function $h$ if for all $d$ and $d'$ in the domain $D$, $(d \equiv_h d') \Rightarrow (d \models f \Leftrightarrow d' \models f)$. Moreover, $\mathrm{Atoms}(\varphi)$ respects $h$ if for all $f \in \mathrm{Atoms}(\varphi)$, $f$ respects $h$.

**Lemma 2.4.1** If $\mathrm{Atoms}(\varphi)$ respects an abstraction function $h$, then the following holds:

*(i)* $d \equiv_h d' \Rightarrow L(d) = L(d')$.

*(ii)* $h(d) = \widehat{d} \Rightarrow L_h(\widehat{d}) = L(d)$.

**Proof** **(i)** Since $\mathrm{Atoms}(\varphi)$ respects $h$, $(d \equiv_h d') \Rightarrow (d \models f \Leftrightarrow d' \models f)$ for all $f \in \mathrm{Atoms}(\varphi)$. According to the definition, $L(d) = \{f \in \mathrm{Atoms}(\varphi) \mid d \models f\}$, Therefore, $L(d) = L(d')$.

**(ii)** The labeling of an abstract state is defined as $L_h(\widehat{d}) = \bigcup_{h(d)=\widehat{d}} L(d)$. In $(i)$, we prove that $d \equiv_h d' \to L(d) = L(d')$. Therefore, we have $L_h(\widehat{d}) = L(d)$. $\square$

Let $\widehat{d}$ be an abstract state. The $h^{-1}(\widehat{d})$ denotes the set of states which are mapped to $\widehat{d}$ by $h$, i.e., $h^{-1}(\widehat{d}) = \{d \mid h(d) = \widehat{d}\}$. $L_h(\widehat{d})$ is *consistent*, if all concrete states corresponding to $\widehat{d}$ satisfy all labels in $L_h(\widehat{d})$, i.e., for all $d \in h^{-1}(\widehat{d})$ it holds that $d \models \bigwedge_{f \in L_h(\widehat{d})} f$.

**Theorem 2.4.1** *Let $h$ be an abstraction function and $\varphi$ be an* $\mathrm{ACTL}^\star$ *specification where the atomic subformulas respect $h$. Then the following holds:*

*(i)* $L_h(\widehat{d})$ *is consistent for all abstract states $\widehat{d}$ in $M_h$;*

*(ii)* $M \preceq M_h$*, i.e.,* $M_h \models \varphi \;\Rightarrow\; M \models \varphi$.

**Proof** **(i)** By Lemma 2.4.1, for all $h(d) = \widehat{d}$, $L_h(\widehat{d}) = L(d)$. By definition, $d \models \bigwedge_{f \in L(d)} f$. Therefore, $L_h(\widehat{d})$ is consistent.

**(ii)** According to the definition of $M_h$, the set of atomic formulas for $M_h$ is $A_t$ which is also the set of atomic formulas for $M$. According to Lemma 2.4.1, $L_h(\widehat{s}) = L(s) = L(s) \cap A_t$ where $h(s) = \widehat{s}$. In the definition of existential abstraction,

$$R_h(\widehat{s}, \widehat{s'}) = \exists s, s'[h(s) = \widehat{s} \wedge h(s') = \widehat{s'} \wedge R(s, s')].$$

By definition, $(s, s') \in R$ implies that $(\widehat{s}, \widehat{s'}) \in R_h$ and $s \in I$ implies that $\widehat{s} \in I_h$ because $I_h(\widehat{s}) = \exists s[h(s) = \widehat{s} \wedge I(s)]$. If we define the relation $H = \{(s, \widehat{s}) \mid \widehat{s} = h(s)\}$, then $M \preceq M_h$ according to the definition of the simulation relation. $\square$

In other words, correctness of the abstract model implies correctness of the concrete model.

**Theorem 2.4.2** *Given an abstraction function $h$ and its corresponding auto-abstraction function $\mathcal{H}$, then $M_h \cong M_{\mathcal{H}}$ where $M_{\mathcal{H}}$ is the abstract structure corresponding to the auto-abstraction function $\mathcal{H}$.*

**Proof** Assume that $\mathcal{I} : \widehat{D} \rightarrow Img(\mathcal{H})$ is a function defined by $\mathcal{I}(h(d)) = rep([d])$. First, we will show that $\mathcal{I}$ is well-defined and that $\mathcal{I}$ is a bijection. Second, using $\mathcal{I}$ we will build a bijection between the states of $M_h$ and $M_{\mathcal{H}}$.

From the definition, $h(d_1) = h(d_2)$ implies that $rep([d_1]) = rep([d_2])$ which in turn implies that $\mathcal{I}(h(d_1)) = \mathcal{I}(h(d_2))$. Therefore, $\mathcal{I}$ is a well defined function. If $d_1 \in Img(\mathcal{H})$, then there exists a $d_2 \in D$, where $d_1 = rep([d_2])$. Moreover, $\mathcal{I}(h(d_2)) = rep(d_2) = d_1$, so $\mathcal{I}$ is a surjection. On the other hand, if $\mathcal{I}(h(d_1)) = \mathcal{I}(h(d_2))$, then $rep([d_1]) = rep([d_2])$ which implies that $h(d_1) = h(d_2)$. Hence $\mathcal{I}$ is an injection. Since $\mathcal{I}$ is injective and surjective, $\mathcal{I}$ is a bijection.

$$
\begin{tikzcd}
M_h \arrow[r, leftrightarrow, "\cong"] & M_{\mathcal{H}} \\
\widehat{s} \arrow[u, phantom, sloped, "\in"] \arrow[r, leftrightarrow, "\mathcal{I}"] & \mathcal{I}(\widehat{s}) \arrow[u, phantom, sloped, "\in"]
\end{tikzcd}
$$

Figure 2.4: Commuting diagram between $M_h$ and $M_{\mathcal{H}}$

As defined before, $S = D$ is the set of states of $M$; $S_h = \widehat{D}$ is the set of states of $M_h$; and $S_{\mathcal{H}} = Img(\mathcal{H})$ is the set of states of $M_{\mathcal{H}}$. Let $\mathcal{I}(I_h) = \{\mathcal{I}(\widehat{s}) \mid \widehat{s} \in I_h\}$. Next we will show that $\mathcal{I}(I_h) = I_{\mathcal{H}}$, i.e., the bijection $\mathcal{I}$ preserves the initial states. Consider an arbitrary state $\widehat{s_0} \in I_h$. Since $\widehat{s_0} \in I_h$, there exists a state $s_0 \in S$ such that $h(s_0) = \widehat{s_0}$ and $s_0 \in I$. Since $\mathcal{H}(s_0) = rep([s_0]) = \mathcal{I}(h(s_0)) = \mathcal{I}(\widehat{s_0})$, and $I_{\mathcal{H}}$ is the existential abstraction of $I$, it follows that $\mathcal{I}(\widehat{s_0}) \in I_{\mathcal{H}}$.

The proof for the transition relation is very similar. Following the same convention, we have

$$\mathcal{I}(R_h) = \{(\mathcal{I}(\widehat{s}), \mathcal{I}(\widehat{s'})) \mid (\widehat{s}, \widehat{s'}) \in R_h\}.$$

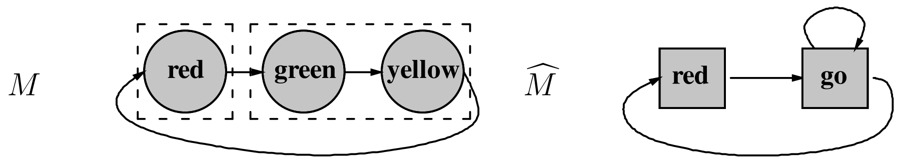



Figure 2.5: Abstraction of a Traffic Light.

Therefore, $\mathcal{I}(I_\mathcal{H}) \subseteq I_\mathcal{H}$ and $\mathcal{I}(R_h) \subseteq R_\mathcal{H}$. Since $I$ is a bijection, the argument given above holds in the reverse direction. Thus, $\mathcal{I}(I_h) = I_\mathcal{H}$ and $\mathcal{I}(R_h) = R_\mathcal{H}$. The relation of the structures is shown in Figure 2.4. This proves that $M_h \cong M_\mathcal{H}$. $\Box$

However, if the abstract model invalidates an $\mathrm{ACTL}^\star$ specification, i.e., $M_h \not\models \varphi$, *the actual model may still satisfy the specification.*

**Example 2.4.1** Assume that for a traffic light controller (see Figure 2.5), we want to prove $\psi = \mathbf{AG\,AF}(state = red)$ using the abstraction function $h(red) = red$ and $h(green) = h(yellow) = go$. It is easy to see that $M \models \psi$ while $M_h \not\models \psi$. There exists an infinite trace $\langle red, go, go, \ldots \rangle$ that invalidates the specification.

If an abstract counterexample does not correspond to any concrete counterexample, we call it *spurious*. For example, $\langle red, go, go, \ldots \rangle$ in the above example is a spurious counterexample.

When the set of possible states is given as the product $D_1 \times \cdots \times D_n$ of smaller domains, it is usually easy to generate abstraction functions for each domain, i.e., $h_i : D_i \rightarrow \widehat{D_i}$. In this scenario, an abstraction $h$ can be described as an n-tuple $(h_1(d_1), \ldots, h_n(d_n))$ where $\widehat{D}$ is equal to $\widehat{D_1} \times \cdots \widehat{D_n}$. We write $h = (h_1, \ldots, h_n)$. The equivalence relations $\equiv_i$ corresponding to the individual surjections $h_i$ induce an equivalence relation $\equiv_h$ over the entire domain $D =$

$D_1 \times \cdots \times D_n$ in the obvious manner:

$$(d_1, \cdots, d_n) \equiv_h (e_1, \cdots, e_n) \text{ iff } d_1 \equiv_1 e_1 \wedge \cdots \wedge d_n \equiv_n e_n$$

## 2.5 Approximation for existential abstraction

It is usually computationally expensive to compute existential abstraction directly [75]. Instead of building $M_h$ directly, approximation is often used to reduce the complexity. If a Kripke structure $\tilde{M} = (S_h, \tilde{I}, \tilde{R}, L_h)$ satisfies

1. $I_h \subseteq \tilde{I}$ and
2. $R_h \subseteq \tilde{R}$

then we say that $\tilde{M}$ *approximates* $M_h$. Intuitively, if $\tilde{M}$ approximates $M_h$, then $\tilde{M}$ is *more* abstract than $M_h$, i.e., has more behaviors than $M_h$. (In the terminology of Chapter 1, $\tilde{M}$ is an over-approximation.)

**Theorem 2.5.1** *Let $h$ be an abstraction function and $\varphi$ be an* ACTL$^\star$ *specification where the atomic subformulas respect $h$. Then $\tilde{M}$ simulates $M_h$, i.e., $M_h \preceq \tilde{M}$.*

Since $\preceq$ is a preorder, $M \preceq M_h \preceq \tilde{M}$ according to Theorem 2.4.1 and Theorem 2.5.1. In [31], Clarke, Grumberg and Long define a practical transformation $\mathcal{T}$ which applies the existential abstraction operation directly to variables at the innermost level of the formula. This transformation generates a new structure $\tilde{M}_{\mathcal{T}}$ as follows. Assume that $R = R_1 \wedge \ldots \wedge R_n$ where each $R_i$ defines the transition relation for a single variable. Then, we apply abstraction to each $R_i$ separately, i.e.,

$$\mathcal{T}(R) = (R_1)_h \wedge \ldots \wedge (R_n)_h.$$

and analogously for $I$. Finally, $\tilde{M}_{\mathcal{T}}$ is given by $(S_h, \mathcal{T}(I), \mathcal{T}(R), L_h)$. As a simple example consider a system $M$ which is a synchronous composition of two systems $M_1$ and $M_2$, or in other words $M = M_1 \| M_2$. Both $M_1$ and $M_2$ define the transition of one variable. In this case $\tilde{M}_{\mathcal{T}}$ is equal to $(M_1)_h \| (M_2)_h$. Note that the existential abstraction operation is applied to each process individually. Since $\mathcal{T}$ is applied at the innermost level, abstraction can be performed before building the BDDs for the transition relation. This abstraction technique is usually fast and easy to implement. However, it has potential limitations in checking certain properties. Since $\tilde{M}_{\mathcal{T}}$ is a coarse abstraction, there exist many properties which cannot be checked on $\tilde{M}_{\mathcal{T}}$ but can still be verified using a finer approximation. The following small example will highlight some of these problems.

**Example 2.5.1** A Kripke structure for a sensor-based traffic light example is shown in Figure 2.6. It includes two finite state machines (FSMs), one for a traffic light and one for an automobile. The traffic light $M_t$ has four states $\{$*red, green1, green2, yellow*$\}$, and the automobile $M_a$ also has four states $\{$*stop1, stop2, drive, slow*$\}$. $M_t$ starts in the state *red*. When it senses that the automobile has waited for some time (in state *stop2*), it triggers a transition to state *green1* which allows the automobiles to move. $M_a$ starts from state *stop1* and transitions according to the states of $M_t$. The safety property we want to prove is that when the traffic light is red, the automobile should either slow down or stop. This can be written in ACTL as follows:

$$\varphi \;\equiv\; \mathbf{AG}[\neg(State_t = red \wedge State_a = drive)]$$

The composed machine $M_t \parallel M_a$ is shown in Figure 2.6. It is easy to see that the property $\varphi$ is true. Let us assume that we want to collapse states $\{$*green1, green2, yellow*$\}$ into one state *go*. If we use the transformation $\mathcal{T}$, which applies abstraction before we compose $M_t$ and $M_a$, property $\varphi$ does not hold as indicated by the shaded state in $abs(M_t) \parallel abs(M_a)$. On the other hand, if we apply this abstraction *after* composing $M_t$ and $M_a$, states *(green2, drive)* and *(yellow, drive)* are collapsed into one state($abs(M_t \parallel M_a)$), and the property $\varphi$ still holds. Thus, by abstracting the individual components and then composing we introduce too many spurious behaviors.

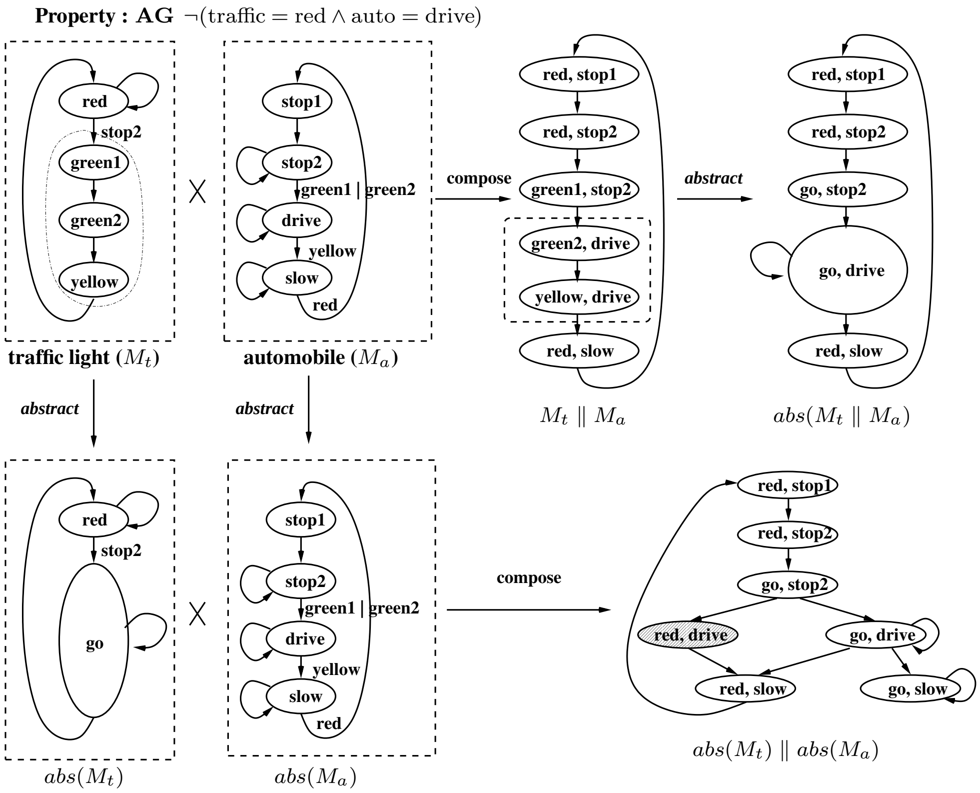



Figure 2.6: Traffic light example

It is desirable to obtain an approximation structure $\tilde{M}$ which is more precise than the structure $\tilde{M}_{\mathcal{T}}$ obtained by the technique proposed in [31]. All the transitions in the abstract structure $M_h$ are included in both $\tilde{M}$ and $\tilde{M}_{\mathcal{T}}$. Note that the state sets of $M_h$, $\tilde{M}$ and $\tilde{M}_{\mathcal{T}}$ are the same and $M \preceq M_h \preceq \tilde{M}_{\mathcal{T}}$. In summary, $M_h$ is intended to be built but is computationally expensive. $\tilde{M}_{\mathcal{T}}$ is easy to build but extra behaviors are added into the structure. Our aim is to build a model $\tilde{M}$ which is computationally easier but a more refined approximation of $M_h$, i.e.

$$M \preceq M_h \preceq \tilde{M} \preceq \tilde{M}_{\mathcal{I}}.$$

## 2.6 Remaining Problems

Existential abstraction is a framework to apply abstraction in model checking. However, there are several important problems which remain unsolved.

- Human interaction is required to provide abstraction functions. This process usually requires great creativity and experience. In many cases, this is impractical. Therefore, generating abstraction functions automatically is very important for verification.

- Existential abstraction is not a complete approach, i.e. $M_h \not\models \varphi$ may not imply that $M \not\models \varphi$ (see Example 2.4.1). Therefore, when the abstraction cannot verify a property, refinement is required. Currently, no refinement algorithms have been developed.

- As discussed before, approximation introduces many spurious transitions (see Example 2.5.1). How to improve the approximation accuracy is unknown.

- In some cases, building BDDs for abstraction functions is a hard problem. It is not known how to build the abstract structure in such cases.

In this thesis, we will address all these problems. We will describe a counterexample-guided abstraction refinement methodology in Chapter 3. In Chapter 4, we extend our refinement algorithms to full **ACTL**. In Chapter 5, we show how to reduce abstraction overhead using abstract BDDs.

# Chapter 3

# Counterexample-guided Abstraction Refinement

In this chapter, we describe an automatic abstraction technique for ACTL$^\star$ specifications which is based on an analysis of the structure of formulas appearing in the program. In general, our technique computes an upper approximation of the original program. Thus, when a specification is true in the abstract model, it will also be true in the concrete design. However, if the specification is false in the abstract model, the counterexample may be the result of some behavior in the approximation which is not present in the original model. When this happens, it is necessary to refine the abstraction so that the behavior which caused the erroneous counterexample is eliminated. The main contribution of this work is an efficient automatic refinement technique which uses information obtained from erroneous counterexamples. The refinement algorithm keeps the size of the abstract state space small due to the use of abstraction functions which distinguish many degrees of abstraction for each program variable. Practical experiments including a large Fujitsu IP core design confirm the competitiveness of our implementation [25].

The methodology is shown in Figure 3.1. Given a program $P$ and a property $\varphi$, the first step is to generate initial abstraction functions and build the initial abstract Kripke structure $\widehat{M}$ accordingly. The traditional model checker will check if $\varphi$ holds on $\widehat{M}$. If not, it will generate a counterexample. The next step checks if the counterexample is spurious or not. If it spurious, the final refinement step will refine the abstraction and redo the whole process.

In the following, we will describe detailed algorithms for each steps.

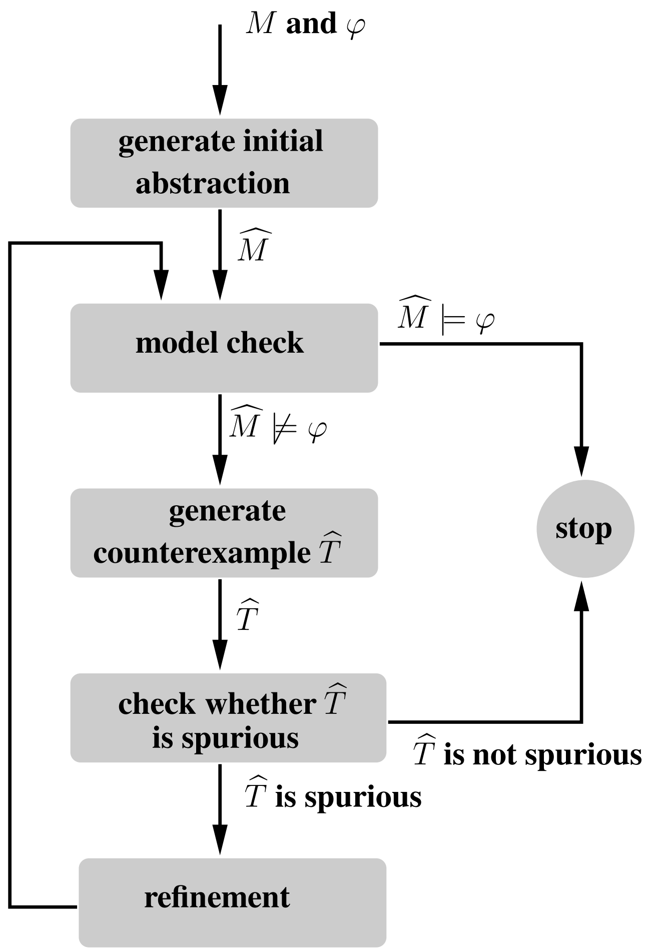



Figure 3.1: Counterexample-guided abstraction refinement methodology

## 3.1 Generating the initial abstraction

Assume that we are given a program $P$ with $n$ variables $\{v_1, \cdots, v_n\}$. $A_t$ is a set of atomic formulas obtained from $P$. As an example, $A_t$ can be $\mathrm{Atoms}(P)$.

Given an atomic formula $f \in A_t$, let $var(f)$ be the set of variables appearing in $f$, e.g., $var(x = y)$ is $\{x, y\}$. Given a set of atomic formulas $U$, $var(U)$ equals $\bigcup_{f \in U} var(f)$. In general, for any syntactic entity $X$, $var(X)$ will be the set of variables appearing in $X$. We say that two atomic formulas $f_1$ and $f_2$ *interfere* if and only if $var(f_1) \cap var(f_2) \neq \emptyset$. Let $\equiv_I$ be the relation on $A_t$ that is the reflexive, transitive closure of the interference relation. According to the following lemma, $\equiv_I$ is an equivalence relation.

**Lemma 3.1.1** *$\equiv_I$ is symmetric, reflexive and transitive. Hence, $\equiv_I$ is an equivalence relation.*

**Proof** $\equiv_I$ satisfies the following properties:

- symmetric: trivially, $f_1 \equiv_I f_2$ implies that $f_2 \equiv_I f_1$;
- reflexive: since $var(f) \cap var(f) \neq \emptyset$, therefore, $f \equiv_I f$;
- transitive: assume that $f_1 \equiv_I f_2$ and $f_2 \equiv_I f_3$. Since $\equiv_I$ is the transitive closure of interfere relation. Therefore, $f_1 \equiv_I f_3$.

Overall, $\equiv_I$ is an equivalence relation. $\square$

The equivalence class of an atomic formula $f \in A_t$ is called the *formula cluster* of $f$ and is denoted by $[f]$. Let $f_1$ and $f_2$ be two atomic formulas. Then $var(f_1) \cap var(f_2) \neq \emptyset$ implies that $[f_1] = [f_2]$. In other words, a variable $v_i$ cannot appear in formulas that belong to two different formula clusters according to the following lemma.

**Lemma 3.1.2** *If $[f_1] \neq [f_2]$, then $var([f_1]) \cap var([f_2]) = \emptyset$.*

**Proof** Assume that $var([f_1]) \cap var([f_2]) \neq \emptyset$. According to the definition of $\equiv_I$, $f_1 \equiv_I f_2$. Since $\equiv_I$ is equivalence relation, $[f_1] = [f_2]$. This contradicts the condition. Therefore, $var([f_1]) \cap var([f_2]) = \emptyset$. $\square$

Moreover, the formula clusters induce an equivalence relation $\equiv_V$ on the set of variables $V$ in the following way:

> $v_i \equiv_V v_j$ if and only if $v_i$ and $v_j$ appear in atomic formulas that belong to the same formula cluster.

The equivalence classes of $\equiv_V$ are called *variable clusters*. For instance, consider a formula cluster $FC_i = \{v_1 > 3, v_1 = v_2\}$. The corresponding variable cluster is $VC_i = \{v_1, v_2\}$. Let $\{FC_1, \ldots, FC_m\}$ be the set of formula clusters and $\{VC_1, \ldots, VC_m\}$ the set of corresponding variable clusters. In another word,

$$A_t = \bigcup_{i=1}^{m} FC_i,\ FC_i \cap FC_j = \emptyset\ (i \neq j)$$

and

$$var(A_t) = \bigcup_{i=1}^{m} VC_i, VC_i = var(FC_i),\ VC_i \cap VC_j = \emptyset\ (i \neq j).$$

We construct the initial abstraction $h = (h_1, \ldots, h_m)$ as follows. For each $h_i$, we set $D_{VC_i} = \prod_{v \in VC_i} D_v$, i.e., $D_{VC_i}$ is the domain corresponding to the variable cluster $VC_i$. Since the variable clusters form a partition of the set of variables $V$, it follows that $D = D_{VC_1} \times \cdots \times D_{VC_m}$. For each variable cluster $VC_i = \{v_{i_1}, \ldots, v_{i_k}\}$, the corresponding abstraction $h_i$ is defined on $D_{VC_i}$ as follows.

> $h_i(d_1, \cdots, d_k) = h_i(e_1, \cdots, e_k)$ if and only if for all atomic formulas $f \in FC_i$, $(d_1, \cdots, d_k) \models f \Leftrightarrow (e_1, \cdots, e_k) \models f$.

In other words two values are in the same equivalence class if they cannot be "distinguished" by atomic formulas appearing in the formula cluster $FC_i$. The following example illustrates how we construct the initial abstraction $h$.

```
init(x) := 0;                    init(y) := 1;
next(x) := case                  next(y) := case
    reset = TRUE : 0;                reset = TRUE : 0;
    x < y : x + 1;                   (x = y) ∧ ¬(y = 2) : y + 1;
    x = y : 0;                       (x = y) : 0;
    else : x;                        else : y;
esac;                            esac;
```

Figure 3.2: Transition block for Example 3.1.1

**Example 3.1.1** Consider the program $P$ with three variables $x, y \in \{0, 1, 2\}$, and $reset \in \{\text{TRUE}, \text{FALSE}\}$ shown in Figure 3.2. The set of atomic formulas is $A_t = \{(reset = \text{TRUE}), (x = y), (x < y), (y = 2)\}$. There are two formula clusters, $FC_1 = \{(x = y), (x < y), (y = 2)\}$ and $FC_2 = \{(reset = \text{TRUE})\}$. The corresponding variable clusters are $\{x, y\}$ and $\{reset\}$, respectively. Consider the formula cluster $FC_1$. Values $(0, 0)$ and $(1, 1)$ are in the same equivalence class because for all the atomic formulas $f$ in the formula cluster $FC_1$ it holds that $(0, 0) \models f$ if and only if $(1, 1) \models f$. It can be shown that the domain $\{0, 1, 2\} \times \{0, 1, 2\}$ is partitioned into a total of five equivalence classes by this criterion. We denote these classes by the natural numbers $0, 1, 2, 3, 4$, and list them below:

$$
\begin{aligned}
&1 = \{(0, 0), (1, 1)\},\\
&2 = \{(0, 1)\},\\
&3 = \{(0, 2), (1, 2)\},\\
&4 = \{(1, 0), (2, 0), (2, 1)\},\\
&5 = \{(2, 2)\}
\end{aligned}
$$

The domain $\{\text{TRUE}, \text{FALSE}\}$ has two equivalence classes – one containing FALSE and the other TRUE. Therefore, we define two abstraction functions $h_1 : \{0, 1, 2\}^2 \rightarrow \{0, 1, 2, 3, 4\}$ and $h_2 : \{\text{TRUE}, \text{FALSE}\} \rightarrow$

$\{\text{TRUE}, \text{FALSE}\}$. The first function $h_1$ is given by

$$
\begin{array}{l}
h_1(0,0) = h_1(1,1) = 0 \\
h_1(0,1) = 1 \\
h_1(0,2) = h_1(1,2) = 2 \\
h_1(1,0) = h_1(2,0) = h_1(2,1) = 3 \\
h_1(2,2) = 4
\end{array}
$$

The second function $h_2$ is just the identity function, i.e., $h_2(reset) = reset$.

On the other hand, let us assume that we choose $A_t = \{(reset = \text{TRUE}), (x = y), (x < y), (y = 2), (x = 0), (y = 0)\}$. Then we will have $FC_1 = \{(x = y), (x < y), (y = 2), (x = 0), (y = 0)\}$ and $FC_2 = \{(reset = \text{TRUE})\}$. As mentioned before, there are two abstraction functions which are defined by

$$
\begin{array}{l}
h_1(0,0) = 0 \\
h_1(0,1) = 1 \\
h_1(0,2) = 2 \\
h_1(1,1) = 3 \\
h_1(1,2) = 4 \\
h_1(1,0) = h_1(2,0) = 5 \\
h_1(2,1) = 6 \\
h_1(2,2) = 7 \\
h_2(reset) = reset
\end{array}
$$

Apparently, the abstraction functions are different from the first example.

From the above example, it is easy to see that choosing different set of atomic formulas results in different abstraction functions. When the number of atomic formulas are small, the obtained abstract model is usually small. When the sufficient number of atomic formulas are selected, the obtained abstract model will be isomorphic to the original model, e.g. when the abstraction function is the identity function.

In our experience, $\text{Atoms}(P)$ or subsets of $\text{Atoms}(P)$ are good choices for $A_t$.

## 3.2 Model checking the abstract model

Given an $\mathrm{ACTL}^\star$ specification $\varphi$, an abstraction function $h$ (assume that $\varphi$ respects $h$), and a program $P$ with a finite set of variables $V = \{v_1, \cdots, v_n\}$, let $\widehat{M}$ be the abstract Kripke structure corresponding to the abstraction function $h$. We use standard symbolic model checking procedures to determine whether $\widehat{M}$ satisfies the specification $\varphi$. If it does, then by Theorem 2.4.1 we can conclude that the original Kripke structure also satisfies $\varphi$. Otherwise, assume that the model checker produces a counterexample $\widehat{T}$ corresponding to the abstract model $\widehat{M}$. In the remainder of this section, we will focus on counterexamples which are either *(finite) paths* or *loops*.

### 3.2.1 Identification of spurious path counterexamples

First, we will tackle the case when the counterexample $\widehat{T}$ is a path $\langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$. The following example highlights the case where the abstract counterexample $\widehat{T}$ does correspond to some concrete counterexample.

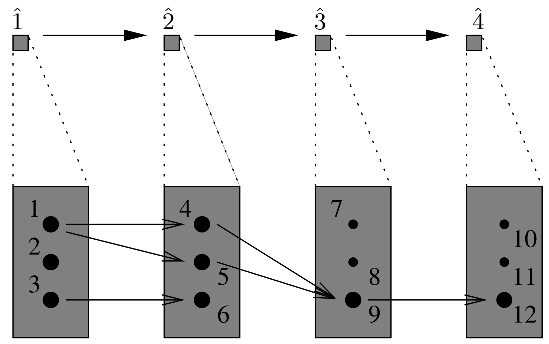



Figure 3.3: Abstract counterexample corresponds to a real trace

**Example 3.2.1** Consider a program with only one variable with domain $D = \{1, \cdots, 12\}$. Assume that the abstraction function $h$ maps $x \in D$ to $\lfloor (x - 1)/3 \rfloor + 1$. There are four abstract states corresponding to the equivalence classes $\{1,2,3\}$, $\{4,5,6\}$, $\{7,8,9\}$, and $\{10,11,12\}$. We call these abstract states $\widehat{1}$, $\widehat{2}$, $\widehat{3}$, and $\widehat{4}$. The transitions between states in the concrete model are indicated by the arrows in Figure 3.3. Small dots denote non-reachable states. Suppose that we obtain an abstract counterexample $\widehat{T} = \langle \widehat{1}, \widehat{2}, \widehat{3}, \widehat{4} \rangle$. It is easy to see that $\widehat{T}$ corresponds to some real trace. As an example, $1 \rightarrow 5 \rightarrow 9 \rightarrow 12$. Therefore, we say that $\widehat{T}$ is not spurious.

Given an abstract state $\widehat{s}$, the set of concrete states $s$ such that $h(s) = \widehat{s}$ is denoted by $h^{-1}(\widehat{s})$, i.e., $h^{-1}(\widehat{s}) = \{s | h(s) = \widehat{s}\}$. We extend $h^{-1}$ to sequences in the following way: $h^{-1}(\widehat{T})$ is the set of concrete paths given by the following expression

$$\{\langle s_1, \cdots, s_n \rangle | \bigwedge_{i=1}^{n} h(s_i) = \widehat{s_i} \wedge I(s_1) \wedge \bigwedge_{i=1}^{n-1} R(s_i, s_{i+1})\}.$$

We will occasionally write $h^{-1}_{\text{path}}$ to emphasize the fact that $h^{-1}$ is applied to a sequence. Next, we give a *symbolic* algorithm to compute $h^{-1}(\widehat{T})$. Let $S_1 = h^{-1}(\widehat{s_1}) \cap I$ and $R$ be the transition relation corresponding to the unabstracted Kripke structure $M$. For $1 < i \leq n$, we define $S_i$ in the following manner: $S_i := Img(S_{i-1}, R) \cap h^{-1}(\widehat{s_i})$. In the definition of $S_i$, $Img(S_{i-1}, R)$ is the forward image of $S_{i-1}$ with respect to the transition relation $R$. The sequence of sets $S_i$ is computed symbolically using OBDDs and the standard image computation algorithm. The following lemma establishes the correctness of this procedure.

**Lemma 3.2.1** *The following are equivalent:*

*(i) The path $\widehat{T}$ corresponds to a concrete counterexample.*

*(ii) The set of concrete paths $h^{-1}(\widehat{T})$ is non-empty.*

*(iii) For all $1 \leq i \leq n$, $S_i \neq \emptyset$.*

**Proof** We first prove that $(i)$ and $(ii)$ are equivalent, then prove that $(ii)$ and $(iii)$ are equivalent.

$(i) \Rightarrow (ii)$ Assume that $\widehat{T}$ corresponds to a concrete counterexample $T = \langle s_1, \ldots, s_n \rangle$. From the definition of $\widehat{T}$, $h(s_i) = \widehat{s_i}$ and $s_i \in h^{-1}(\widehat{s_i})$. Since $T$ is a trace in the concrete model, it has to satisfy the transition relation and start from initial state, i.e. $R(s_i, s_{i+1})$ and $s_1 \in I$. From definition of $h^{-1}(\widehat{T})$, $T \in h^{-1}(\widehat{T})$.

$(ii) \Rightarrow (i)$ Assume that $h^{-1}(\widehat{T})$ is non-empty, we pick a trace $\langle s_1, \ldots, s_n \rangle$ from $h^{-1}(\widehat{T})$. According to the definition of $h^{-1}(\widehat{T})$, $h(s_i) = \widehat{s_i}$. Therefore, $\widehat{T}$ corresponds to a concrete counterexample.

$(ii) \Rightarrow (iii)$ Assume that $h^{-1}(\widehat{T})$ is not empty, then there exists a path $\langle s_1, \ldots, s_n \rangle$ where $h(s_i) = \widehat{s_i}$ and $s_1 \in I$. Therefore, we have $s_1 \in S_1$. Let us assume that $S_i \neq \emptyset$ and $s_i \in S_i$. From the definition of $h^{-1}(\widehat{T})$, $s_{i+1} \in Img(s_i, R)$ and $s_{i+1} \in h^{-1}(\widehat{s_{i+1}})$. Therefore, $s_{i+1} \in Img(s_i, R) \cup h^{-1}(\widehat{s_{i+1}})$. It is easy to see that

$$s_i \in S_i \rightarrow Img(s_i, R) \subseteq Img(S_i, R)$$

is a tautology. Therefore, $s_{i+1} \in Img(S_i, R) \cup h^{-1}(\widehat{s_{i+1}})$. According to the algorithm, $S_{i+1} = Img(S_i, R) \cap h^{-1}(\widehat{s_{i+1}})$. Therefore, $s_{i+1} \in S_{i+1}$ and $S_{i+1} \neq \emptyset$. By induction, $S_i \neq \emptyset$, for all $0 < i \leq n$.

$(iii) \Rightarrow (ii)$ For the other direction assume that $S_i \neq \emptyset$ for $1 \leq i \leq n$. We choose a state $s_n \in S_n$ and construct a trace backward inductively. Assume that $s_i \in S_i$, from definition of $S_i$, we have that $s_i \in Img(S_{i-1}, R) \cap h^{-1}(\widehat{s_i})$ and $S_{i-1}$ is not empty. Select $s_{i-1}$ from $S_{i-1}$. From the definition of $S_{i-1}$, $S_{i-1} \subseteq h^{-1}(\widehat{s_{i-1}})$. Therefore, $s_{i-1} \in h^{-1}(\widehat{s_{i-1}})$. By induction, $s_1 \in S_1 = h^{-1}(\widehat{s_1}) \cap I$. Therefore, the trace $\langle s_1, \ldots, s_n \rangle$ that we have constructed satisfies the definition of $h^{-1}(\widehat{T})$. So $h^{-1}(\widehat{T})$ is not empty. $\square$

Suppose that condition (iii) of Lemma 3.2.1 is violated, and let $i$ be the largest index such that $S_i \neq \emptyset$. Then $\widehat{s_i}$ is called the *failure state* of the spurious counterexample $\widehat{T}$.

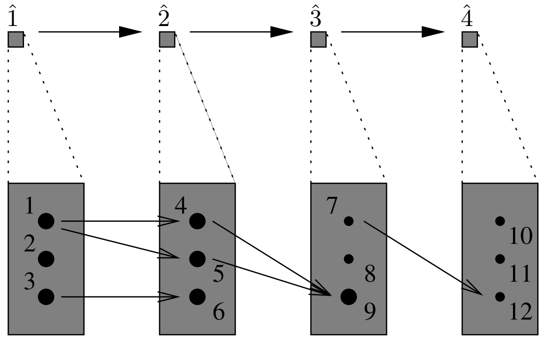



Figure 3.4: An abstract counterexample

**Algorithm SplitPATH**($\widehat{T}$)
(S, n) = Reach_Restrict($\widehat{T}$)
**if** $n = 0$ **then** output "counterexample exists"
**else** output n, S

Figure 3.5: SplitPATH checks a spurious abstract path.

**Example 3.2.2** Consider the similar program to Example 3.2.1 with one variable with domain $D = \{1, \cdots, 12\}$. Also assume that the abstraction function $h = \lfloor (x-1)/3 \rfloor + 1$. We call these abstract states $\widehat{1}$, $\widehat{2}$, $\widehat{3}$, and $\widehat{4}$. The transitions between states in the concrete model are indicated by the arrows in Figure 3.4; small dots denote non-reachable states. Note that the transition between 9 and 12 in Example 3.2.1 is redirected between 7 and 12. It is easy to see that $\widehat{T}$ is spurious. Using the terminology of Lemma 3.2.1, we have

$S_1 = \{1,2,3\}$, $S_2 = \{4,5,6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Notice that $S_4$ and therefore $Img(S_3, R)$ are both empty. Thus, $\widehat{s_3}$ is the failure state.

**function Reach_Restrict**$(\widehat{T})$

$S = h^{-1}(\widehat{s_1}) \cap I$
$j = 1$
**while** $(S \neq \emptyset$ and $j < n)$ $\{$
    $j = j + 1$
    $S_{\text{prev}} = S$
    $S = Img(S, R) \cap h^{-1}(\widehat{s_j})$ $\}$
**if** $S \neq \emptyset$ **then return** $(S, 0)$
**else return** $(S_{\text{prev}}, j)$

Figure 3.6: Reach_Restrict computes the reachable states within $\widehat{T}$.

It follows from Lemma 3.2.1 that if $h^{-1}(\widehat{T})$ is empty (i.e., if the counterexample $\widehat{T}$ is spurious), then there exists a minimal $i$ $(2 \leq i \leq n)$ such that $S_i = \emptyset$. The symbolic Algorithm **SplitPATH** in Figure 3.2.1 computes this number and the set of states $S_{i-1}$; the states in $S_{i-1}$ are called *dead-end* states. After the detection of the dead-end states, we proceed to the refinement step (see Chapter 3.3). On the other hand, if the conditions stated in Lemma 3.2.1 are true, then **SplitPATH** will report a "real" counterexample and we can stop.

### 3.2.2 Identification of spurious loop counterexamples

Now we consider the case when the counterexample $\widehat{T}$ is a loop, which we write as $\langle \widehat{s_1}, \ldots, \widehat{s_n} \rangle^\omega$. Note that the general loop counterexample may start with a path and then end with a loop, i.e., $\widehat{T} = \langle \widehat{s_1}, \ldots, \widehat{s_j} \rangle \langle \widehat{s_{j+1}}, \ldots, \widehat{s_n} \rangle^\omega$. We focus on discussing the first case. However, the derived lemmas and theorems will apply for the general cases as well. It is easy to see that $\widehat{T}$ can also be written as a infinite path, i.e.,

$$\widehat{T} = \langle \widehat{s_1} \ldots \widehat{s_n}, \widehat{s_1} \ldots \widehat{s_n} \ldots \rangle.$$

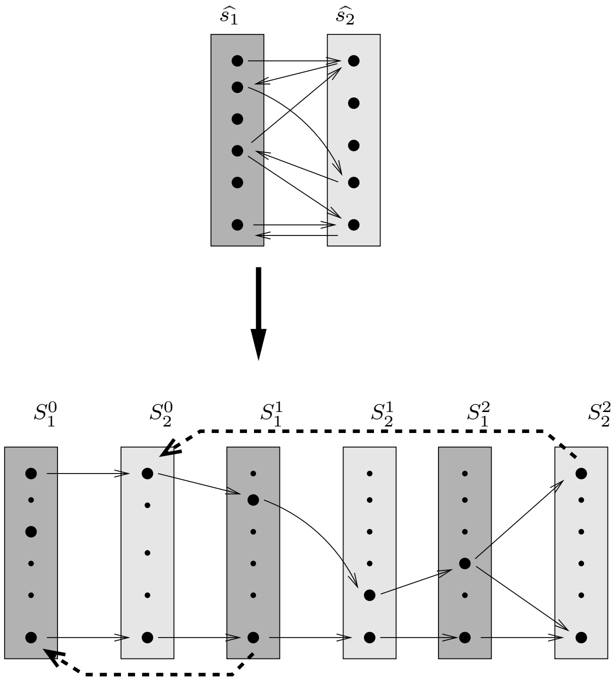



Figure 3.7: A loop counterexample, and its unwinding.

Intuitively, $\widehat{T}$ corresponds to a concrete infinite path $T = \langle s_1 \dots s_j \dots \rangle$ where $R(s_j, s_{j+1})$ if and only if for all $j \geq 1$, $s_j \in h^{-1}(\widehat{s}_{ind(j)})$ where $ind(j) = (j-1) \bmod n + 1$. Since this case is more complicated than the path counterexamples, we first present an example in which some of the typical situations occur.

**Example 3.2.3** We consider a loop $\langle \widehat{s_1}\widehat{s_2} \rangle^\omega$ as shown in Figure 3.7. In order to find out if the abstract loop corresponds to concrete loops, we unwind the counterexample as demonstrated in the figure. There are two situations where cycles occur. In the figure, for each of these situations, an example cycle (the first one occurring) is indicated by a fat dashed arrow. We make the following important observations:

*(i)* A given abstract loop may correspond to several concrete loops of *different size*.

*(ii)* Each of these loops may start at different stages of the unwinding.

*(iii)* The unwinding eventually becomes periodic (in our case $S_2^0 = S_2^2$), but only after several stages of the unwinding. The size of the period is the least common multiple of the size of the individual loops, and thus, in general *exponential*.

We conclude from the example that a naive algorithm may have exponential time complexity due to an exponential number of loop unwindings. The following theorem however shows that a polynomial number of unwindings is sufficient. Let $min$ be the minimum size of all abstract states in the loop, i.e., $min = \min_{1 \leq j \leq n} |h^{-1}(\widehat{s_j})|$. $\widehat{T}_{\text{unwind}}$ denotes the the finite abstract path $\langle \widehat{s_1}, \ldots, \widehat{s_n} \rangle^{min+1}$, i.e., the path obtained by unwinding the loop part of $\widehat{T}$ $(min + 1)$ times.

**Theorem 3.2.1** *The following are equivalent:*

*(i)* $\widehat{T}$ *corresponds to a concrete counterexample.*

*(ii)* $h_{\text{path}}^{-1}(\widehat{T}_{\text{unwind}})$ *is not empty.*

**Proof** Let us first start with some easy observations. Recall that $R$ is the transition relation of the Kripke structure. By definition, the elements of $h_{\text{path}}^{-1}(\widehat{T}_{\text{unwind}})$ have the following form

$$\langle b_1^1, \ldots, b_n^1, \quad \ldots, \quad b_1^{min+1}, \ldots, b_n^{min+1} \rangle \qquad (*)$$

for which the following property hold:

$$b_j^k \in h^{-1}(\widehat{s_j}) \text{ for all } b_j^k \text{ in } P.$$

Each such path $P$ has length $L := n \times (min + 1)$, and we can equivalently write $P$ in the form

$$\langle d_1, \ldots, d_L \rangle \qquad (**)$$

with the properties

1. $d_1 \in h^{-1}(\widehat{s_1}) \cap I$, and
2. for all $j \le L$, if $d_j \in h^{-1}(\widehat{s}_{ind(j)})$ then $d_{j+1} \in h^{-1}(\widehat{s}_{ind(j+1)})$.

Recall that $min$ was defined to be the size of the smallest abstract state in the loop, i.e., $\min\{|h^{-1}(\widehat{s_1})|, \ldots, |h^{-1}(\widehat{s_n})|\}$, and let $M$ be the index of an abstract state $\widehat{s_M}$ s.t. $|h^{-1}(\widehat{s_M})| = min$. (Such a state must exist, because the minimum must be obtained somewhere.)

$(i) \Rightarrow (ii)$ Suppose there exists a concrete counterexample. Since the counterexample contains a loop, there exists an *infinite path* $I = \langle c_1, \ldots \rangle$ such that $c_1 \in h^{-1}(\widehat{s_1})$, and for all $j$, if $c_j \in h^{-1}(\widehat{s}_{ind(j)})$, then $c_{j+1} \in h^{-1}(\widehat{s}_{ind(j+1)})$. According to $(**)$, the finite prefix $\langle c_1, \ldots, c_L \rangle$ of $I$ is contained in $h^{-1}_{\text{path}}(\widehat{T}_{\text{unwind}})$, and thus $h^{-1}_{\text{path}}(\widehat{T}_{\text{unwind}})$ is not empty.

$(ii) \Rightarrow (i)$ Suppose that $h^{-1}_{\text{path}}(\widehat{T}_{\text{unwind}})$ contains a finite path $P$.

**Claim:** *There exists a state which appears at least twice in $P$.*

**Proof of Claim:** Suppose $P$ is in form $(*)$. Consider the states $b_M^1, b_M^2, \ldots, b_M^{min+1}$. By $(*)$, all $b_M^k$ are contained in $h^{-1}(\widehat{s_M})$. By definition of $M$, however, $h^{-1}(\widehat{s_M})$ contains only $min$ elements, and thus there must be at least one repetition in the sequence $b_M^1, b_M^2, \ldots, b_M^{min+1}$. Therefore, there exists a repetition in the finite path $P$, and the claim is proved. $\Box$ (Claim)

Let us now write $P$ in form $(**)$, i.e., $P = \langle d_1, \ldots, d_L \rangle$, and let a repetition be given by two indices $\alpha < \beta$, s.t. $d_\alpha = d_\beta$. Because of the repetition, there must be a transition from $d_{\beta-1}$ to $d_\alpha$, and therefore, $d_\alpha$ is the successor state of $d_{\beta-1}$ in a cycle. We conclude that

$$\langle d_1, \ldots, d_{\alpha-1}\rangle\langle d_\alpha, \ldots, d_{\beta-1}\rangle^\omega$$

is a concrete counterexample. □

We conclude that loop counterexamples can be reduced to path counterexamples. In Figure 3.8, we describe the algorithm **SplitLOOP** which is an extension of **SplitPATH**. In the algorithm, $\widehat{T}_{\text{unwind}}$ is computed by the subprogram **unwind**.

**Algorithm SplitLOOP**($\widehat{T}$)
$min = \min\{|h^{-1}(\widehat{s_1}) \cap I|, \ldots, |h^{-1}(\widehat{s_n})|\}$
$\widehat{T}_{\text{unwind}} = \mathbf{unwind}(\widehat{T}, min + 1)$
Compute $j$ and $S_{\text{prev}}$ as in **SplitPATH**($\widehat{T}_{\text{unwind}}$)
$k := ind(j)$
$p := ind(j + 1)$
**output** $S_{\text{prev}}, k, p$

Figure 3.8: SplitLOOP checks if an abstract loop is spurious

If the abstract counterexample is spurious, then the algorithm **SplitLOOP** outputs a set $S_{\text{prev}}$ and indices $k, p$, such that the following conditions hold:

1. The states in $S_{\text{prev}}$ correspond to the abstract state $\widehat{s_p}$, i.e., $S_{\text{prev}} \subseteq h^{-1}(\widehat{s_p})$
2. All states in $S_{\text{prev}}$ are reachable from $h^{-1}(\widehat{s_1}) \cap I$.
3. $k$ is the successor index of $p$ within the loop, i.e., if $p = n$ then $k = 1$, and otherwise $k = p + 1$.
4. There is no transition from a state in $S_{\text{prev}}$ to $h^{-1}(\widehat{s_k})$, i.e., $Img(S_{\text{prev}}, R) \cap h^{-1}(\widehat{s_k})$ is empty.

5. Therefore, $\widehat{s_p}$ is the failure state of the loop counterexample.

Thus, the final situation encountered is indeed very similar as in the case of path counterexamples. Note that the nontrivial feature of the algorithm **Split-LOOP** is the fact that $min$ unwindings of the loop are necessary. The correctness of this approach is not trivial, and details are deferred to the appendix.

There may be cases where $min$ is a large number. Unwinding the loop may be impossible. In the following, we describe a new practical technique which use fixpoint computation to check whether a loop counterexample is spurious.

Given a Kripke structure $M = (S, I, R, L)$, and an abstract trace $\widehat{T} = \langle\widehat{s_1}, \ldots, \widehat{s_n}\rangle^{\omega}$, assume that $Q = \bigcup_{j=1}^{n} Q_j$ be a set of states where $Q_j$ is a subset of states such that

$$Q_j \cap Q_k = \emptyset, |Q_1| = |h^{-1}(\widehat{s_1}) \cap I|, \text{ and } |Q_j| = |h^{-1}(s_j)| \text{ for } 1 < j \leq n.$$

Assume that $\{\rho_1, \ldots, \rho_n\}$ be a set of arbitrary **bijection** functions where $\rho_1 : Q_1 \rightarrow (h^{-1}(s_1) \cap I)$ and $\rho_j : Q_j \rightarrow h^{-1}(s_j)$ for $1 < j \leq n$. Then we define $g : Q \rightarrow S$ be a function where for $q \in Q_j$,

$$g(q) = \rho_j(q).$$

It is easy to see that $g$ is a well-defined function. We define a new Kripke structure $N = (S^N, I^N, R^N, L^N)$, where $S^N = Q$, $I^N = Q_1$, and

$$\begin{array}{rcl} R^N & = & \{(q, q') \mid R(g(q), g(q')), q \in Q_n, q' \in Q_1\} \\ & \cup & \{(q, q') \mid R(g(q), g(q')), q \in Q_j, q' \in Q_{j+1} \text{ for } 1 \leq j < n\} \end{array}$$

Intuitively, $N$ only captures the transitions occurring between $h^{-1}(\widehat{s_j})$ and $h^{-1}(\widehat{s}_{ind(j+1)})$. Assume that

$$T^N = \langle q_1 \ldots q_i\rangle\langle q_{i+1} \ldots q_m\rangle^{\omega}$$

be a loop trace in $N$, i.e., $q_j \in S^N$ for $1 \leq j \leq m$. Then the following lemma holds.

**Lemma 3.2.2** *The following claims are valid:*

*(i) If* $q_1, q_2 \in Q_k$*, then* $(q_1, q_2) \notin R^N$*;*

*(ii)* $(m - i) \bmod n = 0$*, i.e., the period of* $T^N$ *is the multiple of* $n$*.*

**Proof** $(i)$ This can be directly derived from the definition of $R^N$.

$(ii)$ Assume that $m - i = \alpha * n + \beta$ where $\beta \neq 0$. Assume that $q_{i+1} \in Q_\eta$ and $q_m \in Q_\xi$. Then there exists $k$,

$$m - i = k * n + (n - \eta + 1) + \xi = \alpha * n + \beta$$

or

$$(n - \eta + 1) + x = \beta \bmod n$$

Note that $(q_m, q_{i+1}) \in R^N$, and $q_{i+1} \in Q_\eta$. Then we have

$$\xi = \begin{cases} \eta - 1 & \eta \neq n \\ 1 & \eta = n \end{cases}$$

For either cases, $(n - \eta + 1) + \xi = 0 \mod n$. This contradict the hypothesis that $\beta \neq 0$. Therefore, $(ii)$ claim is correct. $\square$

Furthermore, the following theorem holds.

**Theorem 3.2.2** $\widehat{T}$ *corresponds to a concrete loop counterexample if and only if* $N, I^N \models \mathbf{EG}\,\texttt{true}$.

**Proof** Assume that $N, I^N \models \mathbf{EG}\,\texttt{true}$, then there exists an infinite path on $N$. Since $N$ is finite, the infinite path must forms a loop. Assume that this loop is

$$T^N = \langle q_1 \dots q_i \rangle \langle q_{i+1} \dots q_m \rangle^\omega$$

where $q_j \in S^N$ for $1 \leq j \leq m$. Apparently,

$$T = \langle g(q_1) \dots g(q_i) \rangle \langle g(q_{i+1}) \dots g(q_m) \rangle^{\omega}$$

is an infinite trace on $M$ since $g(q_j) \in S$ for all $1 \leq j \leq n$. In the following, we will argue that $T$ is the concrete loop counterexample corresponding to $\widehat{T}$, i.e., we need to prove that

$$\widehat{T} = \langle \widehat{s_1} \dots \widehat{s_n} \rangle^{\omega}$$

corresponds to the infinite path

$$T = \langle g(q_1) \dots g(q_{i+1}) \dots g(q_m), g(q_{i+1}) \dots g(q_m), \dots \rangle.$$

Since the period of $T$ is the multiple of the period of $\widehat{T}$. It is sufficient to show that the finite path

$$T' = \langle g(q_1) \dots g(q_{i+1}) \dots g(q_m) \rangle$$

corresponds to a prefix of $\widehat{T}$ with the same length. According to the definition of $R^N$, $R^N(q_j, q_{j+1})$ implies that $q_j \in Q_{ind(j)}$ and $q_{j+1} \in Q_{ind(j+1)}$. Therefore, for $j \leq m$, $g(q_j) = \rho_{ind(j)}(q_j) \in h^{-1}(\widehat{s}_{ind(j)})$. Therefore, $\widehat{T}$ corresponds to a concrete loop counterexample $T$.

On the other hand, assume that $\widehat{T}$ corresponds to a concrete loop counterexample

$$T = \langle s_1, \dots, s_i \rangle \langle s_{i+1}, \dots, s_m \rangle^{\omega}.$$

According to Lemma 3.2.2, $m - i \bmod n = 0$ and $s_j \in \widehat{s}_{ind(j)}$. Since $\rho_j$ is a bijection, there must exist a set of states $\{q_1, \dots, q_i, \dots, q_m\} \in Q$ where $\rho_{ind(j)}(q_j) = s_j$. Or, $g(q_j) = s_j$. Note that $q_j \in Q_{ind(j)}$. According to the definition of $R^N$, $(q_j, q_{j+1}) \in R^N$ if and only if $R(g(q_j), g(q_{j+1}))$. Therefore, $R(s_j, s_{j+1})$ implies that $R^N(q_j, q_{j+1})$. Also, $R(s_m, s_{j+1})$ implies that

$R^N(q_m, q_{j+1})$. Overall, $T^N = \langle q_1, \ldots, q_j \rangle \langle q_{j+1}, \ldots, q_m \rangle^\omega$ is an infinite trace on $N$. Therefore, $N, I^N \models \mathbf{EG}\,\texttt{true}$.

□

In the implementation, it may be expensive to build $N$ directly. Instead, we build the transition relation $R^N$ on-the-fly during computing $\mathbf{EG}\,\texttt{true}$.

## 3.3 Refining the abstraction

First, we will consider the case when the counterexample $\widehat{T} = \langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$ is a path. Let us return to a previous example for a closer investigation of failure states.

**Example 3.3.1** Recall that in the spurious counterexample of Figure 3.4, the abstract state $\widehat{3}$ was the **failure state**. There are three types of concrete states in the failure state $\widehat{3}$:

*(i)* The **dead-end state** $9$ is reachable, but there are no outgoing transitions to the next state in the counterexample.

*(ii)* The **bad state** 7 is not reachable but outgoing transitions cause the spurious counterexample. The spurious counterexamples is caused by the bad state.

*(iii)* The **irrelevant state** $8$ is neither reachable nor bad.

The goal of the refinement methodology described in this section is to refine $h$ so that the dead-end states and bad states do not belong to *the same* abstract state. Then the spurious counterexample will be eliminated.

If $\widehat{T}$ does not correspond to a real counterexample, by Lemma 3.2.1 (iii) there always exists a set $S_i$ of dead-end states, i.e., $S_i \subseteq h^{-1}(\widehat{s_i})$ with $1 \leq i < n$

such that $Img(S_i, R) \cap h^{-1}(\widehat{s_{i+1}}) = \emptyset$ and $S_i$ is reachable from initial state set $h^{-1}(\widehat{s_1}) \cap I$. Moreover, the set $S_i$ of dead-end states can be obtained as the output $S_{\text{prev}}$ of **SplitPATH** or **SplitLOOP**. Since there is a transition from $\widehat{s_i}$ to $\widehat{s_{i+1}}$ in the abstract model, there is at least one transition from a *bad* state in $h^{-1}(\widehat{s_i})$ to a state in $h^{-1}(\widehat{s_{i+1}})$ even though there is no transition from $S_i$ to $h^{-1}(\widehat{s_{i+1}})$, and thus the set of bad states is not empty. We partition $h^{-1}(\widehat{s_i})$ into three subsets $S_{i,0}$, $S_{i,1}$, and $S_{i,x}$ as follows:

| *Name* | *Partition* | *Definition* |
|---|---|---|
| dead-end states | $S_{i,0}$ | $S_i$ |
| bad states | $S_{i,1}$ | $\{s \in h^{-1}(\widehat{s_i}) \vert \exists s' \in h^{-1}(\widehat{s_{i+1}}).R(s, s')\}$ |
| irrelevant states | $S_{i,x}$ | $h^{-1}(\widehat{s_i}) \setminus (S_{i,0} \cup S_{i,1})$ |

Intuitively, $S_{i,0}$ denotes the set of dead-end states, i.e., states in $h^{-1}(\widehat{s_i})$ that are reachable from initial states. $S_{i,1}$ denotes the set of bad states,i.e., those states in $h^{-1}(\widehat{s_i})$ that are not reachable from initial states, but have at least one transition to some state in $h^{-1}(\widehat{s_{i+1}})$. The set $S_{i,1}$ cannot be empty since we know that there is a transition from $h^{-1}(\widehat{s_i})$ to $h^{-1}(\widehat{s_{i+1}})$. $S_{i,x}$ denotes the set of irrelevant states, i.e., states that are not reachable from initial states, and do not have a transition to a state in $h^{-1}(\widehat{s_{i+1}})$. Since $S_{i,1}$ is not empty, there is a spurious transition $\widehat{s_i} \rightarrow \widehat{s_{i+1}}$. This causes the spurious counterexample $\widehat{T}$. Hence in order to refine the abstraction $h$ so that the new model does not allow $\widehat{T}$, we need a refined abstraction function which separates the two sets $S_{i,0}$ and $S_{i,1}$, i.e., we need an abstraction function, in which no abstract state simultaneously contains states from $S_{i,0}$ and from $S_{i,1}$.

It is natural to describe the needed refinement in terms of equivalence relations: Recall that $h^{-1}(\widehat{s})$ is an equivalence class of $\equiv$ which has the form $E_1 \times \cdots \times E_m$, where each $E_i$ is an equivalence class of $\equiv_i$. Thus, the refinement $\equiv'$ of $\equiv$ is obtained by partitioning the equivalence classes $E_j$ into subclasses, which amounts to refining the equivalence relations $\equiv_j$. The *size of*

| | 3 | 4 | 5 |
|---|---|---|---|
| 7 | 1 | x | x |
| 8 | 0 | x | 1 |
| 9 | x | 0 | 0 |

Equivalence Class

| | **3/4** | 5 |
|---|---|---|
| 7 | 1 | x |
| 8 | 0 | 1 |
| 9 | 0 | 0 |

Refinement (a)

| | 3 | **4/5** |
|---|---|---|
| **7/9** | 1 | 0 |
| 8 | 0 | 1 |

Refinement (b)

Figure 3.9: Two possible refinements of an Equivalence Class.

*the refinement* is the number of new equivalence classes. Ideally, we would like to find the coarsest refinement that separates the two sets, i.e., the separating refinement with the smallest size.

**Example 3.3.2** *Assume that we have two variables $v_1, v_2$. The failure state corresponds to* one equivalence class $E_1 \times E_2$*, where* $E_1 = \{3, 4, 5\}$ *and* $E_2 = \{7, 8, 9\}$*. In Figure 3.9, dead-end states $S_{i,0}$ are denoted by 0, bad states $S_{i,1}$ by 1, and irrelevant states by $x$.*

*Let us consider two possible partitions of $E_1 \times E_2$ :*

- *Case (a) :* $\{(3, 4), (5)\} \times \{(7), (8), (9)\}$ *(6 classes)*
- *Case (b) :* $\{(3), (4, 5)\} \times \{(7, 9), (8)\}$ *(4 classes)*

*Clearly, case (b) generates a coarser refinement than case (a). It can be easily checked that no other refinement is coarser than (b).*

In general, the problem of finding the coarsest refinement problem is computationally intractable.

**Theorem 3.3.1** *The problem of finding the coarsest refinement is NP-hard.*

The proof is provided in Chapter 3.6.

We therefore need to obtain a good heuristics for abstraction refinement. When $S_{i,x}$ is empty, there is a polynomial algorithm which can find the coarsest refinement. The algorithm **PolyRefine** (see Figure 3.10) corresponds to this case. Let $P_j^+, P_j^-$ be two projection functions, such that for $s = (d_1, \ldots, d_m)$, $P_j^+(s) = d_j$ and $P_j^-(s) = (d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$. Then $proj(S_{i,0}, j, a)$ denotes the *projection* set $\{P_j^-(s) | P_j^+(s) = a, s \in S_{i,0}\}$. Intuitively, the condition $proj(S_{i,0}, j, a) \neq proj(S_{i,0}, j, b)$ in the algorithm means that there exists $(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \in proj(S_{i,0}, j, a)$ and $(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \notin proj(S_{i,0}, j, b)$. According to the definition of $proj(S_{i,0}, j, a)$, $s_1 = (d_1, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in S_{i,0}$ and $s_2 = (d_1, \ldots, d_{j-1}, b, d_{j+1}, \ldots, d_m) \notin S_{i,0}$, i.e., $s_2 \in S_{i,1}$. The only way to separate $s_1$ and $s_2$ into different equivalence classes is that $a$ and $b$ have to be in different equivalence classes of $\equiv'_j$, i.e., $a \not\equiv'_j b$.

**Algorithm PolyRefine**

**for** j := 1 **to** m {
  $\equiv'_j := \equiv_j$
  **for** every $a, b \in E_j$ {
    **if** $proj(S_{i,0}, j, a) \neq proj(S_{i,0}, j, b)$
      **then** $\equiv'_j := \equiv'_j \setminus \{(a, b)\}$ }}

Figure 3.10: The algorithm **PolyRefine**

**Lemma 3.3.1** *When $S_{i,x} = \emptyset$, the relation $\equiv'_j$ computed by* **PolyRefine** *is an equivalence relation which refines $\equiv_j$ and separates $S_{i,0}$ and $S_{i,1}$. Furthermore, the equivalence relation $\equiv'_j$ is the coarsest refinement of $\equiv_j$.*

The proof of this lemma is provided in Chapter 3.6.

Note that in symbolic presentation, the projection operation $proj(S_{i,0}, j, a)$ amounts to computing a generalized cofactor, which can be easily done by standard BDD methods. Given a function $f : D \rightarrow \{0, 1\}$, a generalized cofactor of $f$ with respect to $g = (\bigwedge_{k=p}^{q} x_k = d_k)$ is the function

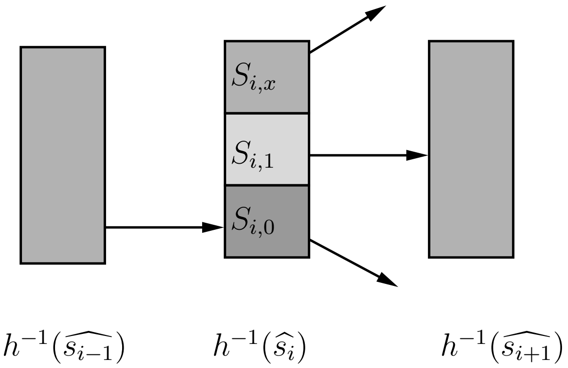



Figure 3.11: Three sets $S_{i,0}, S_{i,1}$, and $S_{i,x}$

$f_g = f(x_1, \ldots, x_{p-1}, d_p, \ldots, d_q, x_{q+1}, \ldots, x_n)$. In other words, $f_g$ is the projection of $f$ with respect to $g$. Symbolically, the set $S_{i,0}$ is represented by a function $f_{S_{i,0}} : D \to \{0,1\}$, and therefore, the projection $proj(S_{i,0}, j, a)$ of $S_{i,0}$ to value $a$ of the $j$th component corresponds to a cofactor of $f_{S_{i,0}}$.

In our implementation, we use an heuristics which is based on the following corollary to the proof of Lemma 3.3.1.

**Corollary 3.3.1** *Even if $S_{i,x}$ is not empty, the relation $\equiv'_j$ computed by* **PolyRefine** *is an equivalence relation which refines $\equiv_j$ and separates $S_{i,0}$ and $S_{i,1}$.*

**Refinement Heuristics** *We merge the states in $S_{i,x}$ into $S_{i,1}$, and use the algorithm* **PolyRefine** *to find the coarsest refinement that separates the sets $S_{i,0}$ and $S_{i,1} \cup S_{i,x}$. The equivalence relation computed by* **PolyRefine** *in this manner is in general not optimal, but it is a correct refinement which separates $S_{i,0}$ and $S_{i,1}$, and eliminates the spurious counterexample. This heuristic has given good results in our practical experiments.*

Since according to Theorem 3.2.1, the algorithm **SplitLOOP** for loop counterexamples works analogously as **SplitPATH**, the refinement procedure for spurious loop counterexamples works analogously, i.e., it uses **SplitLOOP** to identify the failure state, and **PolyRefine** to obtain a heuristic refinement.

Our refinement procedure continues to refine the abstraction function by partitioning equivalence classes until a real counterexample is found, or the $\mathrm{ACTL}^\star$ property is verified. The partitioning procedure is guaranteed to terminate since each equivalence class must contain at least one element. Thus, our method is complete.

**Theorem 3.3.2** *Given a model $M$ and an $\mathrm{ACTL}^\star$ specification $\varphi$ whose counterexample is either path or loop, our algorithm will find a model $\widehat{M}$ such that $\widehat{M} \models \varphi \Leftrightarrow M \models \varphi$.*

**Proof** There are three cases to consider.

*(i)* If $\widehat{M} \models \varphi$, then $M \models \varphi$ according to Theorem 2.4.1

*(ii)* If $\widehat{M} \not\models \varphi$, and the generated abstract counterexample is not spurious, then there exists a concrete counterexample, and hence, $M \not\models \varphi$.

*(iii)* If $\widehat{M} \not\models \varphi$, and the generated abstract counterexample is spurious, then **PolyRefine** will refine the abstraction. Since each refinement step partitions an existing equivalence classes into *strictly* smaller equivalence classes, after a finite number of steps the equivalence relation will become the *equality* relation, and therefore $\widehat{M} = M$. Hence $M \not\models \varphi$.

□

## 3.4 Performance improvements

The symbolic methods described in Chapter 3.2 and Chapter 3.3 can be directly implemented using BDDs. Our implementation uses additional heuristics which are outlined in the following.

### 3.4.1 Detecting more real counterexamples

**Example 3.4.1** Consider a program similar to Example 3.2.1. The program has only one variable $x \in \{1, \cdots, 12\}$. The abstraction function $h(x) = \lfloor (x-1)/3 \rfloor + 1$. There are four abstract states corresponding to the equivalence classes $\{1, 2, 3\}$, $\{4, 5, 6\}$, $\{7, 8, 9\}$, and $\{10, 11, 12\}$. We call these abstract states $\widehat{1}$, $\widehat{2}$, $\widehat{3}$, and $\widehat{4}$. The transitions between states in the concrete model are indicated by the arrows in Figure 3.4.1. Using our **SplitPATH** algorithm, $\widehat{T}$ is spurious. However, there exists a trace $\langle 1, 4, 9, 7, 12 \rangle$ which is a real counterexample. Note that this trace does not correspond to the abstract trace $\widehat{T}$. Instead, it corresponds to an abstract trace $\langle \widehat{1}, \widehat{2}, \widehat{3}, \widehat{3}, \widehat{4} \rangle$.

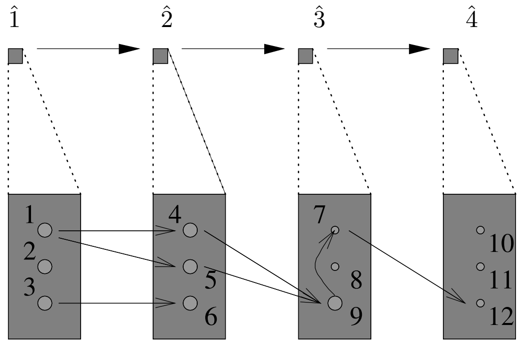



Figure 3.12: Detecting more real counterexamples

Although the trace $\langle 1, 4, 9, 7, 12 \rangle$ does not correspond to the abstract counterexample, finding this trace avoids rechecking the model which may be expensive. Therefore, a relatively small effort to detect such counterexamples is justified as a valuable heuristic. For loop counterexamples, the scenario is similar. Consider the loop counterexample $\widehat{T} = \langle \widehat{1}, \widehat{2} \rangle^{\omega}$ of Figure 3.13. Similar to Example 3.4.1, $\widehat{T}$ is also spurious according to the algorithm **SplitLOOP**. However, there exists an infinite path $\langle 1, 1, \ldots \rangle$ which is a potential counterexample.

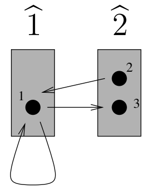



Figure 3.13: A spurious loop counterexample $\langle\widehat{1},\widehat{2}\rangle^\omega$

Given a property $\varphi$ and a Kripke structure $M = (S, I, R, L)$, assume that a general counterexample $\widehat{T}$ includes $n$ abstract states. We modify our original algorithms in order to detect more real counterexamples.

*(i)* We restrict the model to be a smaller model $N = (S^N, I^N, R^N, L^N)$ where the state space $S^N = (\bigcup_{1\leq i\leq n} h^{-1}(\widehat{s_i}))$, $I^N = h^{-1}(\widehat{s_1}) \cap I$, $R^N(s, s') = R(s, s')$ for $s, s' \in S^N$, and $L^N(s) = L(s)$ for $s \in S^N$. Then we check $N, I^N \models \neg\varphi$ is true or not. If a concrete counterexample is found, then the algorithm terminates. We use bounded model checking algorithm [11] to check $N, I^N \models \neg\varphi$. When the problem is hard to decide, we abandon the algorithm and directly go to the second step.

*(ii)* If no counterexample is found, we use **SplitPATH** or **SplitLOOP** to compute a refinement as described above.

This two-phase algorithm is slower than the original one if we do not find a concrete counterexample; in many cases however, it can speed up the search for a concrete counterexample.

### 3.4.2 Abstraction for approximation

Despite the use of partitioned transition relations it is often infeasible to compute the total transition relation of the model $M$ [26]. Therefore, the abstract model $\widehat{M}$ cannot be computed from $M$ directly. In previous work [5, 31], a method which we call *early approximation* has been introduced: first, abstraction is applied to the BDD representation of each transition block and then the BDDs for the partitioned transition relation are built from the already abstracted BDDs for the transition blocks. The disadvantage of early approximation is that it *over-approximates* the abstract model $\widehat{M}$ [27]. In our approach, a heuristic individually determines for each variable cluster $VC_i$, if early approximation should be applied or if the abstraction function should be applied in an exact manner. Our method has the advantage that it balances overapproximation and memory usage. Moreover, the overall method presented in our paper remains complete with this approximation.

**Lemma 3.4.1** *Let $\widehat{R}$ be the abstract transition relation obtained from existential abstraction. Let $\{R_i^{\mathrm{early}}\}$ be a partitioned transition relation obtained from early approximation. Let $\{R_i^{\mathrm{combined}}\}$ be the final partitioned transition relation which we obtain in our approach. Then $\widehat{R} \rightarrow \bigwedge_i R_i^{\mathrm{combined}}$ and $\bigwedge_i R_i^{\mathrm{combined}} \rightarrow \bigwedge_i R_i^{\mathrm{early}}$.*

Thus, the approximation in our approach indeed is intermediate between early approximation and exact existential abstraction. Our method remains complete, because during the symbolic simulation of the counterexample the algorithms **SplitPATH** and **SplitLOOP** treat both forms of overapproximations, i.e., virtual transitions and spurious transitions, in the same way.

### 3.4.3 Abstractions for distant variables

In addition to the methods of Chapter 3.1, we completely abstract variables whose distance from the specification in the *variable dependency graph* is greater than a user-defined constant. Note that the variable dependency graph is also used for this purpose in the localization reduction [5, 68, 72] in a similar way. However, the refinement process of the localization reduction [68] can only turn a completely abstracted variable into a completely unabstracted variable, while our method uses intermediate abstraction functions.

A user-defined integer constant `far` determines which variables are close to the specification $\varphi$. The set NEAR of near variables contains those variables whose distance from the specification in the dependency graph is at most `far`, and $\mathtt{FAR} = var(P) - \mathtt{NEAR}$ is the set of far variables. For variable clusters without far variables, the abstraction function remains unchanged. For variable clusters with far variables their far variables are completely abstracted away, and their near variables remain unabstracted. Note that the initial abstraction for variable clusters with far variables looks similar as in the localization reduction. However, the refinement process of the localization reduction [68] can only turn a completely abstracted variable into a completely unabstracted variable, while our method uses intermediate abstraction functions.

## 3.5 Experimental results

We have implemented our methodology in NuSMV [24] which uses the CUDD package [101] for symbolic representation. We performed two sets of experiments. One set includes four benchmark designs and three industrial designs from Synopsys. The other was performed on an industrial design of a multimedia processor from Fujitsu [1]. All the experiments were carried out on a 200MHz PentiumPro PC with 1GB RAM memory using Linux.

### 3.5.1 Experiments on benchmark circuits

The first benchmark set includes four publicly available designs and three industrial designs. The properties of the design are described in Table 3.1. In the table, the second column (#Var) shows the number of symbolic variables in the design while the third column (#Reg) shows the corresponding number of the

Boolean variables. For example, a symbolic variable with domain whose size equals to eight corresponds to three Boolean variables. Therefore, the number of Boolean variables is always larger or equal to the number of symbolic variables. Overall, thirty seven properties are considered in this benchmark.

| *Design* | #Var | #Reg | #Prop |
|---|---|---|---|
| gigamax | 10 | 16 | 1 |
| guidance | 40 | 55 | 8 |
| waterpress | 6 | 21 | 8 |
| PCI bus | 50 | 89 | 15 |
| ind1 | 72 | 72 | 1 |
| ind2 | 101 | 101 | 1 |
| ind3 | 190 | 190 | 1 |

Table 3.1: Properties of the given benchmark designs

The results for these designs are listed in Table 3.2. Note that average time and space usages per design are reported in this table. In the table, the performance for an enhanced version of NuSMV with cone of influence reduction (**NuSMV + COI**) and our implementation (**NuSMV + ABS**) are compared. #Var and #Prop are properties of the designs: #Var = $x(y)$ means that $x$ is the number of symbolic variables, and $y$ the number of Boolean variables in the design. #Prop is the number of verified properties. The columns #COI and #ABS contain the number of symbolic variables which have been abstracted using the cone of influence reduction (#COI), and our initial abstraction (#ABS). The column "Time" denotes the accumulated running time to verify all #Prop properties of the design. $|TR|$ denotes the maximum number of BDD nodes used for building the transition relation. $|MC|$ denotes the maximum number of *additional* BDD nodes used during the verification of the properties. Thus, $|TR| + |MC|$ is the maximum BDD size during the total model checking process. For the larger examples, we use partitioned transition relations by setting the BDD size limit to 10000.

| *Design* | NuSMV+COI | | | | NuSMV+ABS | | | |
|---|---|---|---|---|---|---|---|---|
| | #COI | Time | $\|TR\|$ | $\|MC\|$ | #ABS | Time | $\|TR\|$ | $\|MC\|$ |
| gigamax | 0 | 0.3 | 8346 | 1822 | 9 | 0.2 | 13151 | 816 |
| guidance | 30 | 35 | 140409 | 30467 | 34-39 | 30 | 147823 | 10670 |
| waterpress | 0-1 | 273 | 34838 | 129595 | 4 | 170 | 38715 | 3335 |
| PCI bus | 4 | 2343 | 121803 | 926443 | 12-13 | 546 | 160129 | 350226 |
| ind1 | 0 | 99 | 241723 | 860399 | 50 | 9 | 302442 | 212922 |
| ind2 | 0 | 486 | 416597 | 2164025 | 84 | 33 | 362738 | 624600 |
| ind3 | 0 | 617 | 584815 | 2386682 | 173 | 15 | 426162 | 364802 |

Table 3.2: Running results for the benchmark designs

On the other hand, we also report the relative time and space difference between our approach and traditional cone of influence reduction in Figure 3.14 and Figure 3.15. In the figures, the x axis corresponds to the number of properties and y axis corresponds to the relative time and space difference respectively (Time(COI)/Time(Abs) and Space(COI)/Space(Abs)). Although our approach uses less than 50% more memory than the traditional cone of influence reduction to *build* the abstract transition relation, it requires one magnitude of memory less during *model checking*. This is an important achievement since the model checking process is the most difficult task in verifying large designs. More significant improvement is further demonstrated by the Fujitsu IP core design.

### 3.5.2 Debugging a multimedia processor

As another example, we verified a multimedia assist (MMA-ASIC) processor developed by Fujitsu [1]. The system configuration of this processor is shown in Figure 3.16 [102]. A dashed line represents a chip boundary. MM-ASIC is connected to a host CPU and external I/O units via "Bus-H", and to SDRAMs via "Bus-R". MM-ASIC consists of a co-processor for multimedia instructions (MMA), a graphic display controller (GDC), peripheral I/O units, and five bus

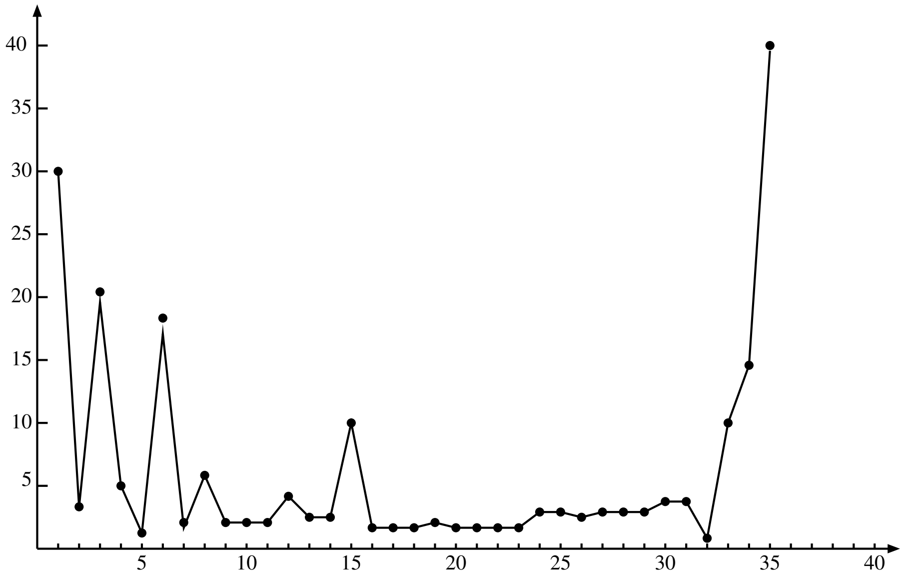



Figure 3.14: The relative time improvement

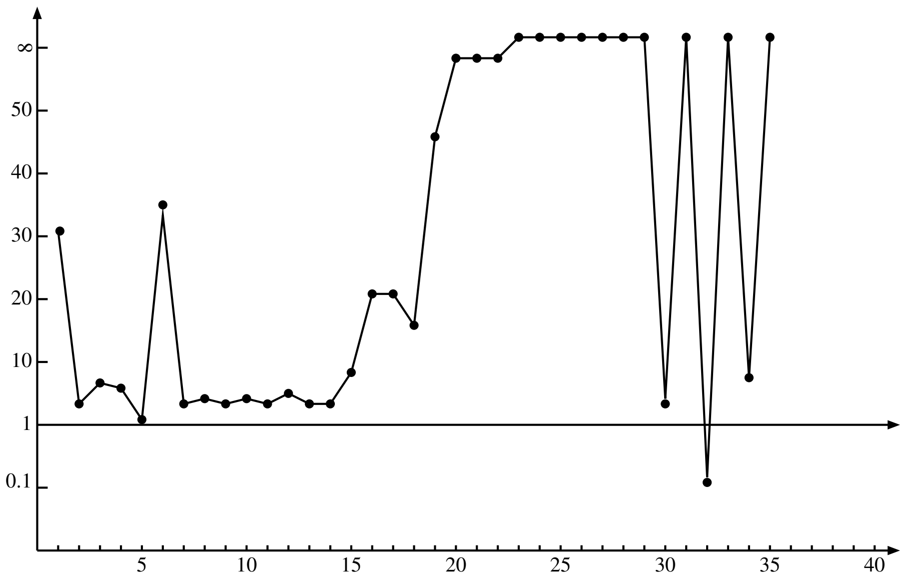



Figure 3.15: The relative time improvement

bridges (BBs).

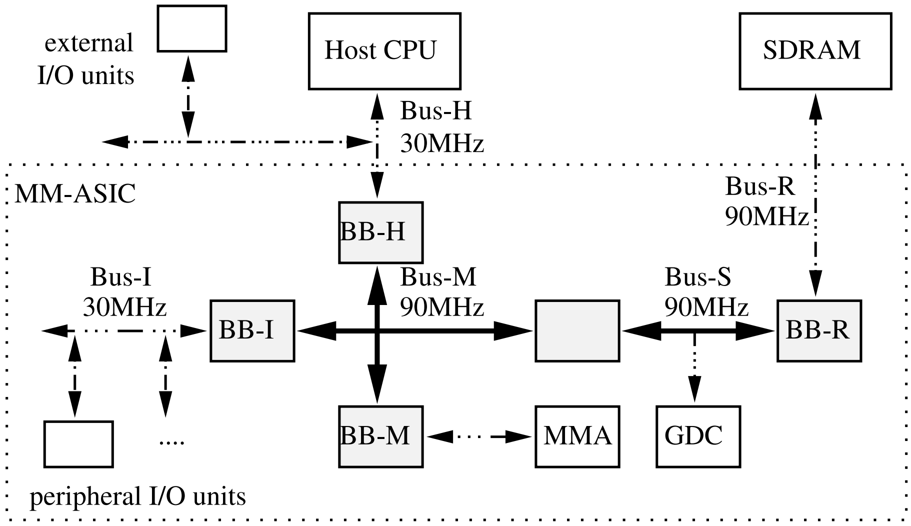



Figure 3.16: Configuration of MMA-ASIC

It is one of the characteristics of a system-on-chip that the design contains bus bridges, because the components of the system may have different interface protocols or different frequencies of operation. Or bus bridges are used to construct a bus hierarchy according to the locality of transactions. MM-ASIC consists of the following five bus bridges.

- ”BB-I” and ”BB-H”: These separate Bus-M from Bus-H and Bus-I, since the bus frequency of Bus-M is different from that of Bus-H and Bus-I.

- ”BB-S”: This separates the transactions between GDC and SDRAM from those between MMA and host CPU, since they are major transactions in MM-ASIC.

- ”BB-R”: This solves the difference of the bus protocols between Bus-R and Bus-S.

- ”Bus-M”: This separates Bus-M from the local bus of MMA.

The RTL implementation of MM-ASIC is described in Verilog-HDL. The total number of lines of code is about 61,500. The verification targets to verify the bus transactions. Therefore, three operational units, peripheral I/Os, MMA, and GDC are omitted. After this elimination of the units, the number of registers is reduced to about 4000. Fujitsu engineers then abstracted away the data path which is not useful for our verification task. The final description contains about 500 latches.

Figure 3.17 shows some control signals and controllers within bus bridges. BB-H, BB-I and BB-M contains a DMA controller ”DMAC” which controls a DMA transfer between SDRAM and a component, such as an external/internal IO and MMA. BB-H contains another controller ”HSTC” which controls a data transfer between a host and the other components but external IOs. BB-R asserts ”FreeS” when it can accept a request from Bus-S. BB-S asserts ”FreeW” (”FreeR”) when it can accept a write (read) request from Bus-M. A bus transaction on Bus-M consists of the following four phases.

- *Arbitration phase* is the first cycle when a bus master asserts a request signal. When more than one master requests, only the master which has the highest priority goes into request phase in the next cycle. ”ReqS”, ”ReqM”, and ”ReqI” are request signals on Bus-M for DMA transfer from/to SDRAM. The signals are asserted by BB-H, BB-M, and BB-I respectively. ”ReqH” is a request asserted by BB-H for normal (non-DMA) data transfer. The priority of the request is $ReqM \prec ReqI \prec ReqS = ReqH$.

- *Request phase* is the next cycle after the arbitration phase. A bus master passes the address and other control signals to a bus slave.

- *Ready phase* is the cycle when the data is ready to be transferred.

”DenO” (”DenI”) is asserted when the write (read) data is ready to transfer in the next cycle. ”Pack” is asserted when the data is transferred between BB-H and a bus bridge, such as BB-M and BB-I, in the next cycle.

- *Transfer phase* is the next cycle after the ready phase. A data is transferred between a master and a slave.

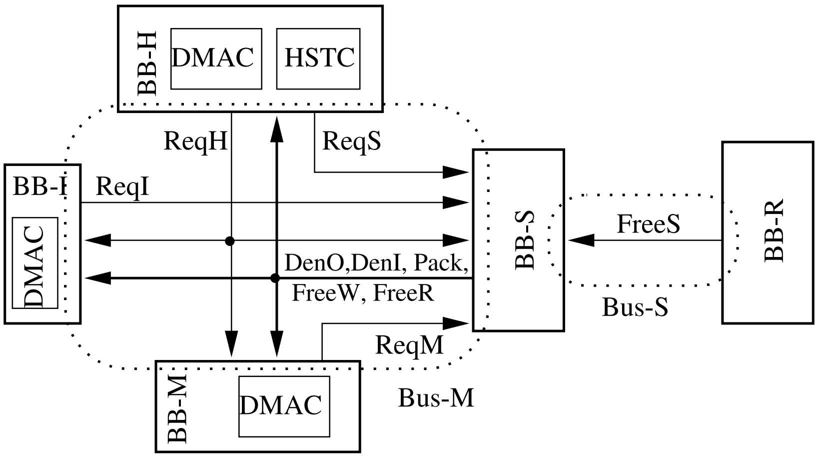



Figure 3.17: Control signals on ”Bus-M”

In [102], the authors verified this design using a ”navigated” model checking algorithm in which state traversal is restricted by navigation conditions provided by the user. Therefore, their methodology is not complete, i.e., it may fail to prove the correctness even if the property is true. Moreover, the navigation conditions are usually not automatically generated. Since our model checker can only accept SMV language, we translated this abstracted Verilog code into 9,500 lines of SMV code.

In order to compare our model checker to others, we tried to verify this design using two state-of-the-art model checkers - Yang’s SMV [106] and NuSMV [24]. We implemented the cone of influence reduction for NuSMV, but not for Yang’s SMV. Both NuSMV+COI and Yang’s SMV failed to verify the design. On the other hand, our system abstracted 144 symbolic variables and with three refinement steps, successfully verified the design, and found a bug which has not been discovered before.

## 3.6 Proofs for Refinement Theorem

Recall that in figure 3.9, we have visualized the special case of two variables and two equivalence relations in terms of matrices:

| | 3 | 4 | 5 |
|---|---|---|---|
| 7 | 1 | x | x |
| 8 | 0 | x | 1 |
| 9 | x | 0 | 0 |

Equivalence Class

| | **3/4** | 5 |
|---|---|---|
| 7 | 1 | x |
| 8 | 0 | 1 |
| 9 | 0 | 0 |

Refinement (a)

| | 3 | **4/5** |
|---|---|---|
| **7/9** | 1 | 0 |
| 8 | 0 | 1 |

Refinement (b)

In order to formally capture this visualization, let us define the **Matrix Squeezing** problem.

**Definition 3.6.1 Matrix Squeezing**

*Given an integer constant* $\Gamma$ *and a finite* $(n, m)$ *matrix with entries* $0, 1, x$, *is it possible to obtain a matrix with* $\leq \Gamma$ *entries by iterating the following operations:*

1. *Merging two compatible rows.*

2. *Merging two compatible columns.*

*Two rows are* compatible*, if there is no position, where one row contains* $1$ *and the other row contains* $0$*. All other combinations are allowed, i.e.,* $x$ *does not affect compatibility.* Merging *two rows means replacing the rows by a new one which contains* $1$ *at those positions where at least one of the two columns contained* $1$*, and* $0$ *at those positions, where at least one of the two columns contained* $0$*.*

*For columns, the definitions are analogous.*

Since **Matrix Squeezing** is a special case of the refinement problem, it is sufficient to show NP-hardness for **Matrix Squeezing**. Then it follows that the refinement problem is NP-hard, too, and thus Theorem 3.3.1 is proved.

As mentioned **Matrix Squeezing** is easy to visualize: If we imagine the symbol $x$ to be transparent, then merging two columns can be thought of as putting the two (transparent) columns on top of each other. **Column Squeezing** is a variant of **Matrix Squeezing**, where only columns can be merged, and the number of rows is left unchanged. We will first show NP-completeness of **Column Squeezing**, and then show NP-completeness of **Matrix Squeezing** by a reduction from **Column Squeezing**.

**Definition 3.6.2 Column Squeezing**

*Given an integer constant* $\Delta$ *and a finite* $(n, m)$ *matrix with entries* $0, 1, x$, *is it possible to obtain a matrix with* $\leq \Delta$ columns *by iterated merging of* columns *?*

The proof will be by reduction from problem GT15 in [48]:

**Definition 3.6.3 Partition Into Cliques**

*Given an undirected graph* $(V, E)$ *and and a number* $K \geq 3$*, is there a partition of* $V$ *into* $k \leq K$ *classes, such that each class induces a clique on* $(V, E)$ *?*

**Theorem 3.6.1 (Karp 72)** *Partition Into Cliques is NP-complete.*

**Theorem 3.6.2 Column Squeezing** *is NP-complete.*

**Proof:** Membership is trivial. Let us consider hardness. We reduce **Partition Into Cliques** to **Column Squeezing**. Given a graph $(V, E)$ and a number $K$, we have to construct a matrix $M$ and a number $\Delta$ such that $M$ can be squeezed to size $\leq \Delta$ iff $(V, E)$ can be partitioned in $\leq K$ cliques.

We construct a $(|V|,|V|)$ matrix $(a_{i,j})$ which is very similar to the adjacency matrix of $(V,E)$:

$$a_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } (i,j) \notin E, i \neq j \\ x & \text{if } (i,j) \in E, i \neq j \end{cases}$$

Assume w.l.o.g. that $V = \{1,\ldots,n\}$. Then it is not hard to see that for all $i,j \in V$, columns $i$ and $j$ are compatible iff $(i,j) \in E$, since the $0$ entries in the matrix were chosen in such a way that the columns corresponding to two non-adjacent edges cannot be merged.

By construction, $(V,E)$ contains a clique $C$ with vertices $c_1,\ldots,c_l$ iff the columns $c_1,\ldots,c_l$ can all be merged into one. (Note however that compatibility is not a transitive relation.)

Thus, $(V,E)$ can be partitioned into $\leq K$ cliques, iff the columns of $(a_{i,j})$ can be merged into $\leq K$ columns. Setting $\Delta = K$ concludes the proof. $\Box$

**Theorem 3.6.3** **Matrix Squeezing** *is NP-complete.*

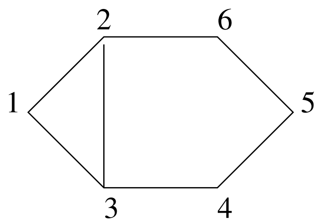
**Proof:** Membership is trivial. We show hardness by reducing **Column Squeezing** to **Matrix Squeezing**. For an integer $n$, let $|bin(n)|$ denote the size of the binary representation of $n$. Given an $(n,m)$ matrix $M$ and a number $\Delta$, it is easy to construct an $(n+1, m+|bin(m-1)|)$ matrix $B(M)$ by adding additional columns to $A$ in such a way that

*(i)* all rows of $B(M)$ become incompatible, and

*(ii)* no new column is compatible with any other (new or old) column.

An easy construction to obtain this is to concatenate the rows of $M$ with the binary encodings of the numbers $0,\ldots,m-1$ over alphabet $\{0,1\}$, such that the $i$th row is concatenated with the binary encoding of the number $i-1$. Since any two different binary encodings are distinguished by at least one position, no two rows are compatible. In addition, we add an $n+1$st row which contains $1$ on positions in the original columns, and $0$ on positions in the new columns. Thus, in matrices of the form $B(M)$, only columns which already appeared in $M$ (with an additional $0$ symbol below) can be compatible.

It remains to determine $\Gamma$. We set $\Gamma := (\Delta + |bin(m-1)|) \times (n+1)$.□ The summand $|bin(m-1)|$ takes into account that we have added $|bin(m-1)|$ columns, and the factor $(n+1)$ takes into account that $\Delta$ is counting columns, while $\Gamma$ is counting matrix entries.□




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | x | x | 0 | 0 | 0 |
| 2 | x | 1 | x | 0 | 0 | x |
| 3 | x | x | 1 | x | 0 | 0 |
| 4 | 0 | 0 | x | 1 | x | 0 |
| 5 | 0 | 0 | 0 | x | 1 | x |
| 6 | 0 | x | 0 | 0 | x | 1 |

**Column Squeezing**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | x | x | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | x | 1 | x | 0 | 0 | x | 0 | 0 | 1 |
| 3 | x | x | 1 | x | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | x | 1 | x | 0 | 0 | 1 | 1 |
| 5 | 0 | 0 | 0 | x | 1 | x | 1 | 0 | 0 |
| 6 | 0 | x | 0 | 0 | x | 1 | 1 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

**Matrix Squeezing**

Figure 3.18: An instance of **Partition into Cliques**, and its reduction images.

**Example 3.6.1** *Figure 3.18 demonstrates how a graph instance is reduced to a matrix instance. Note for example that* $\{1, 2, 3\}$ *is a clique in the graph, and therefore, the columns* $1, 2, 3$ *of the* **Column Squeezing** *problem are compatible. In the* **Matrix Squeezing Instance***, Columns* $7, 8, 9$ *enforce that no rows* *can be merged. Row* $7$ *guarantees that columns* $7, 8, 9$ *can not be merged with columns* $1, \ldots, 6$*.*

In the following, we prove that when $S_{i,x}$ is empty, there exists a polynomial algorithm to find the coarsest refinement. Let $s \in h^{-1}(\widehat{s_i})$ be a state and $P_j^+, P_j^-$ be two projection functions, such that for $s = (d_1, \ldots, d_m)$, $P_j^+(s) = d_j$ and $P_j^-(s) = (d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$. Note that this definition is consistent to the definition in Chapter 3.3. Since $S_{i,x}$ is empty, $S_{i,0}$ and $S_{i,1}$ form a partition of $h^{-1}(\widehat{s_i})$. A refinement of $h^{-1}(\widehat{s_i})$ can be achieved by refining each equivalence relations $\equiv_j$ (and thus, simultaneously, the abstraction functions $h_j$).

We will replace each equivalence relation $\equiv_j$ by the equivalence relation $\equiv'_j$ in the following way: We put two elements $a, b$ of $D_{VC_j}$ in the same equivalence class (symbolically, $a \equiv'_j b$) if and only if the *projection* sets $P_{j,a} = \{P_j^-(s) | P_j^+(s) = a, s \in S_{i,1}\}$ and $P_{j,b} = \{P_j^-(s) | P_j^+(s) = b, s \in S_{i,1}\}$ are equal. Intuitively, this means that any two states which only differ in the $j$th component are either both in $S_{i,1}$ or both not in $S_{i,1}$. As shown in Chapter 2.3, the equivalence relations $\equiv'_j$ $(1 \leq j \leq m)$ define an equivalence relation $\equiv'$ on $D$.

**Lemma 3.3.1** *When* $S_{i,x} = \emptyset$*, the relation* $\equiv'_j$ *computed by* **PolyRefine** *is an equivalence relation which refines* $\equiv_j$ *and separates* $S_{i,0}$ *and* $S_{i,1}$*. Furthermore, the equivalence relation* $\equiv'_j$ *is the coarsest refinement of* $\equiv_j$*.*

**Proof** First, we argue that $\equiv'_j$ is an equivalence relation:

- Reflexivity: for any $a \in E_j$, $(a, a)$ is not removed from $\equiv_j$, therefore, $a \equiv'_j a$;
- Symmetry: $a \equiv'_j b$ implies that $proj(S_{i,0}, j, a) = proj(S_{i,0}, j, b)$. According to **PolyRefine**, $(b, a)$ is not removed from $\equiv_j$. Therefore, $b \equiv'_j a$;

- Transitivity: assume that $a \equiv'_j b$ and $b \equiv'_j c$, Then $proj(S_{i,0}, j, a) = proj(S_{i,0}, j, b)$ and $proj(S_{i,0}, j, b) = proj(S_{i,0}, j, c)$. Hence, $proj(S_{i,0}, j, a) = proj(S_{i,0}, j, c)$. This implies that $a \equiv'_j c$.

Secondly, we show that $\equiv'$ is a correct refinement, i.e., for any two states $s_1 \in S_{i,1}$ and $s_2 \in S_{i,0}$, $s_1 \not\equiv' s_2$. Assume that there are two states $s_1 \in S_{i,1}$ and $s_2 \in S_{i,0}$ where $s_1 \equiv' s_2$. Also assume that $s_1 = (d_1, \ldots, d_m)$ and $s_2 = (e_1, \ldots, e_m)$ where $d_j \equiv'_j e_j$. Without loss of generality, we assume that $d_j \neq e_j$ for $1 \leq j \leq k$ and $d_j = e_j$ for $k < j \leq m$ where $1 < k \leq m$. Consider another state $s_3 = (e_1, d_2, \ldots, d_m)$. Since $e_1 \in E_1$, $d_j \in E_j$ for $1 < j \leq m$, $s_3 \in h^{-1}(\widehat{s_i})$. On the other hand, $s_1 \equiv' s_3$ because $d_1 \equiv'_1 e_1$ and $d_j \equiv'_j d_j$ for all $j$. According to our definition of $\equiv'_1$, any two states which only differ in the $j$th component are either both in $S_{i,1}$ or both not in $S_{i,1}$. Since $s_1 \in S_{i,1}$, it follows that $s_3 \in S_{i,1}$. Furthermore, we consider $s_4 = (e_1, e_2, d_3, \ldots, d_m)$. Following the same argument, $s_3 \equiv' s_4$ and $s_4 \in S_{i,1}$. Therefore, $s_1 \equiv' s_4$. By repeating this step $k$ times, we will obtain that $s_1 \equiv' s_2$ and $s_2 \in S_{i,1}$. Hence, $S_{i,1} \cap S_{i,0} \neq \emptyset$. This contradicts our definition of $S_{i,1}$ and $S_{i,0}$. Therefore, the equivalence relation $\equiv'$ partitions $S_{i,1}$ and $S_{i,0}$ into different equivalence classes.

Finally, we prove that the equivalence relation $\equiv'$ defines the coarsest refinement. Towards contradiction, we assume that there is another equivalence relation $\equiv''$ which defines a coarser refinement than $\equiv'$ and it eliminates the counterexample. Note that a coarser refinement implies that there are a fewer number of equivalence classes generated by $\equiv''$ than $\equiv'$. This implies that there exists a $j$ such that $\equiv''_j$ generates fewer equivalence classes than $\equiv'_j$. Therefore, there must exist two elements $a, b \in D_{VC_j}$ where $a \not\equiv'_j b$ but $a \equiv''_j b$. According to the definition of $\equiv'_j$, $a \not\equiv'_j b$ if and only if there exist two states $s_1$ and $s_2$, s.t. $P_j^+(s_1) = a$, $P_j^+(s_2) = b$ and $P_j^-(s_1) = P_j^-(s_2)$, however, either $s_1 \in S_{i,1} \wedge s_2 \notin S_{i,1}$ or $s_1 \notin S_{i,1} \wedge s_2 \in S_{i,1}$. We will first consider the case of $s_1 \in S_{i,1} \wedge s_2 \notin S_{i,1}$. The second case will follow the same argument. Because $S_{i,x}$ is empty, $s_2 \notin S_{i,1}$ implies that $s_2 \in S_{i,0}$. On the other hand, $a \equiv''_j b$ implies that $s_1 \equiv'' s_2$ according to the definition of $\equiv''$. Therefore, $\equiv''$ cannot partition $S_{i,1}$ and $S_{i,0}$ into different equivalence classes, i.e., it cannot eliminate the counterexample. Hence, $\equiv'$ defines the coarsest refinement. $\square$

# Chapter 4

# Refinement for General ACTL Counterexamples

In the previous chapters, we considered counterexamples of a very simple structure, i.e., paths and loops. Paths and loops have two advantages: (i) they are easy to understand for the human user and facilitate error detection, and (ii) they can be used efficiently in the context of the counterexample-guided abstraction refinement methodology developed in the previous chapter. On the other hand, it is easy to see that such simple counterexamples suffice only for a limited subset of $\mathbf{ACTL}$.

In this chapter, we introduce *tree-like* counterexamples. Tree-like counterexamples retain the abovementioned advantages of path and loop counterexamples, but are complete for $\mathbf{ACTL}$, i.e., whenever an $\mathbf{ACTL}$ specification is violated, a tree-like counterexample can be constructed.

In the first two sections, we explain and motivate the framework for generating tree-like counterexamples. In the last two sections, we describe a symbolic algorithm that generates tree-like counterexamples for all $\mathbf{ACTL}$ formulas and provide a refinement algorithm for such counterexamples.

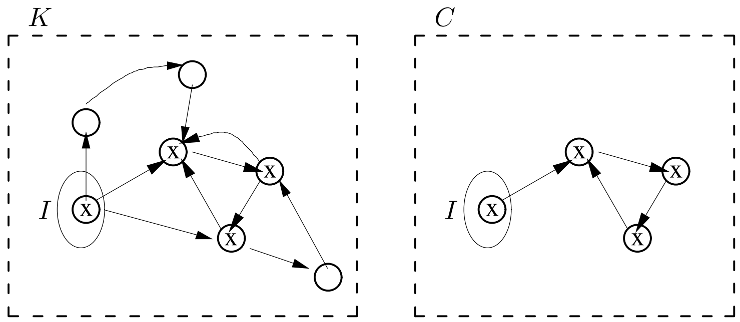


Figure 4.1: An counterexample for $\mathbf{AF}\,\neg x$

## 4.1 What are Counterexamples?

**Definition 4.1.1** Let $K$ be a Kripke structure, and $\varphi$ be a property in a modal logic. A counterexample $C$ is a Kripke structure from which it can be inferred that $K \not\models \varphi$.

While this definition is correct, it is too general to be of practical use, and does not take into account the specifics of temporal logics. The following example highlights what a counterexample looks like in the scenario of $\mathbf{ACTL}$.

**Example 4.1.1** Consider the $\mathbf{ACTL}$ formula $\mathbf{AF}\,\neg x$ on the Kripke structure $K = (S, I, R, L)$ of Figure 4.1. The Kripke Structure $C$ in the same figure corresponds to a trace on $K$ which satisfies $\mathbf{EG}\,x$. Thus, $C$ is a counterexample for $\mathbf{AF}\,\neg x$. By investigating $C$, it is easy to conclude that $\mathbf{AF}\,\neg x$ is false on the original Kripke structure $K$.

Note that the counterexamples for a given Kripke structure can be treated as a kind of "sub-structure". In the above example, $C = (S_C, \{s_C\}, R_C, L_C)$ is a Kripke structure where $S_C \subseteq S$, $s_C \in I$, $R_C \subseteq R$ and $L_C(s) = L(s)$ for $s \in S_C$.

Intuitively, $C$ includes partial behavior of $K$. Recall from Chapter 2.1 that this relationship is expressed by the simulation relation $K \succeq C$, i.e., $K$ simulates $C$. Note that existential properties of the simulated structure $C$ also hold for the simulating structure $K$. (As we know, for universal properties, the converse implication holds, and thus the result follows from the fact that existential properties are negations of universal properties.)

Since $\mathbf{ACTL}$ counterexamples are witnesses for existential properties, we obtain the following definition:

**Definition 4.1.2** Let $K$ be a Kripke structure, and $\varphi$ be an $\mathbf{ACTL}$ property. A counterexample for $\varphi$ is a Kripke structure $C$ such that

1. The counterexample $C$ disproves $\varphi$, i.e., $C \models \neg\varphi$.

2. $K \succeq C$, and therefore, $K \models \neg\varphi$.

Thus, a counterexample is a (typically small) Kripke structure which disproves $\varphi$ in a manner that can be simulated in $K$.

Still, our definition of counterexamples is very general, and allows for complicated counterexamples, as outlined by the following case:

**Example 4.1.2** Let us return to Example 4.1.1 and Figure 4.1 Consider the Kripke structure $C'$ shown in Figure 4.2. It is easy to see that $C'$ is also a counterexample since $K \succeq C'$ and $C' \not\models \mathbf{AF}\,\neg x$. However, $C'$ contains more information than necessary to locate bugs, and is harder to understand. In particular, $C$ contains nested cycles as well as several states and transitions which are not relevant for disproving $\mathbf{AF}\,\neg x$.

Therefore, it is desirable for the user to have $C$ instead of $C'$ as a counterexample.

The problem of understanding counterexamples becomes even more acute for counterexamples of nested properties, where it is hard to find out which part of a counterexample is related to which subformula of the property.

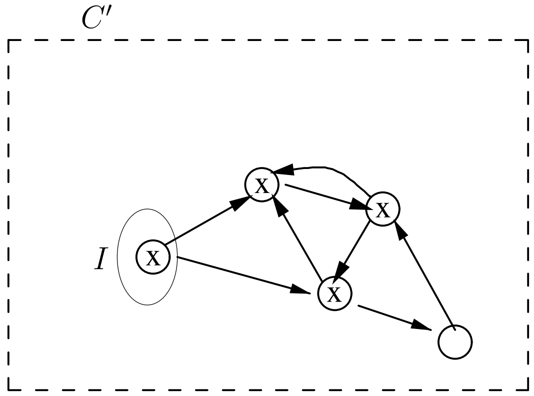



Figure 4.2: A counterexample for $K \models \mathbf{AF}\,\neg x$ in Example 4.1.1

Therefore, it is important for the model checkers to generate counterexamples like $C$ whose structure is simple and easy to analyze. The remainder of this chapter discusses a special type of Kripke structures – *tree-like* Kripke structures which are easy to understand.

## 4.2 Tree-like Kripke structures

Recall that a Kripke structure $K$ is a tuple $(S, I, R, L)$, where $(S, R)$ is a directed graph called the graph of $K$, $I$ is the set of initial states, and $L : S \to 2^{A_t}$ is a labeling function.

Throughout this chapter, we will for simplicity assume that $I = \{s_0\}$ contains a single initial state $s_0$, and that all states in $S$ are reachable from $s_0$. We also assume that the Kripke structure is total. These assumptions simplify the technical exposition. The results can be easily generalized.

In the following, we will first define tree-like Kripke structures. Later, we will show how to build a Kripke structure from finite paths and loops.

**Definition 4.2.1** Given a Kripke structure $K$, the *skeleton* of $K$ is obtained by collapsing all strongly connected components of the graph of $K$ into single nodes. $K$ is *tree-like* if (i) the skeleton of $K$ is a tree and (ii) the strongly connected components of $K$ are directed cycles.

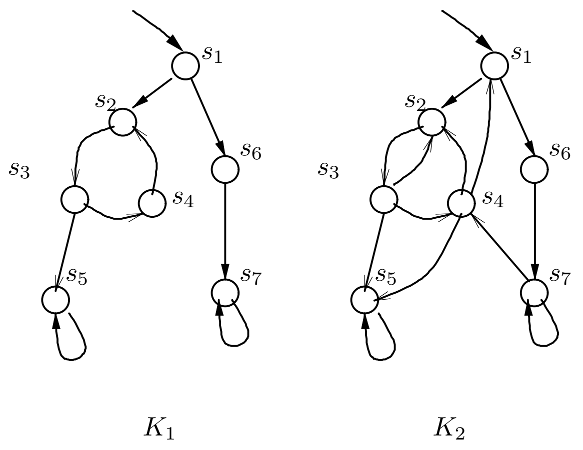



Figure 4.3: The left Kripke structure is tree-like, while the right one is not.

Thus, a tree-like Kripke structure is very similar to its skeleton tree, except for the fact that certain vertices of the skeleton are expanded into cycles.

**Example 4.2.1** In Figure 4.3, $K_1$ is a tree-like Kripke structure while $K_2$ is not a tree-like Kripke structure.

**Lemma 4.2.1** If $K$ is tree-like, then no two distinct directed cycles of $K$ have a common node.

**Proof** If two distinct directed cycles have a common node then their union is contained in one strongly connected component. This contradicts the definition of tree-likeness. $\square$

Thus, tree-like Kripke structures are composed of cycles and connections between them. We will exploit this fact later on in our algorithms. In particular, the following definitions will facilitate the description of natural recursive algorithms.

Given a finite Kripke structure $K$, a *K-path* $p = \langle s_1, \ldots, s_n \rangle$ is the substructure of $K$ whose transition relation is the finite path, i.e., $s_i \in S$ and $(s_i, s_{i+1}) \in R$. A *K-loop* $l = \langle s_1, \ldots, s_n \rangle^\omega$ is the substructure of $K$ where $s_i \in S$, $(s_i, s_{i+1}) \in R$, and $(s_n, s_1) \in R$. Both K-paths and K-loops are called *bricks* . The first state of a brick $q$ is called *anchor*, denoted by $anchor(q)$. Given a brick $q$, $S_q$ stands for the set of states appearing in $q$, and $R_q$ for its transition relation.

**Definition 4.2.2** Given a finite Kripke structure $K = (S, I, R, L)$ and a set of bricks $Q = \{q_1, \ldots, q_n\}$, the *constructed* Kripke structure $K_Q = (S_Q, I_Q, R_Q, L_Q)$ is defined as follows.

- $S_Q = \bigcup_1^n S_{q_i}$, therefore, $S_Q \subseteq S$;
- $I_Q = I \cap S_Q$;
- $R_Q = \bigcup_1^n R_{q_i}$;
- $L_Q : S_Q \rightarrow 2^{A_t}$ and $L_Q(s) = L(s)$.

**Lemma 4.2.2** Given a Kripke structure $K = (S, \{s_0\}, R, L)$ where $|S| > 1$, there exists a set of bricks $Q$ such that the constructed Kripke structure $K_Q$ is the same as $K$.

Note that $K_Q$ is not the substructure induced by $S_Q$, since only transitions from the bricks are allowed in $K_Q$.

Lemma 4.2.2 shows that any Kripke structure can be decomposed as a set of K-paths and K-loops. The following example shows one way to decompose a Kripke structure.

**Example 4.2.2** Consider the Kripke structure shown in Figure 4.4. The initial state is $s_1$. A possible set of bricks $Q$ can be

$$\{\langle s_1, s_2 \rangle, \langle s_1, s_5, s_6 \rangle, \langle s_6 \rangle^\omega, \langle s_2, s_3, s_4 \rangle^\omega\}$$

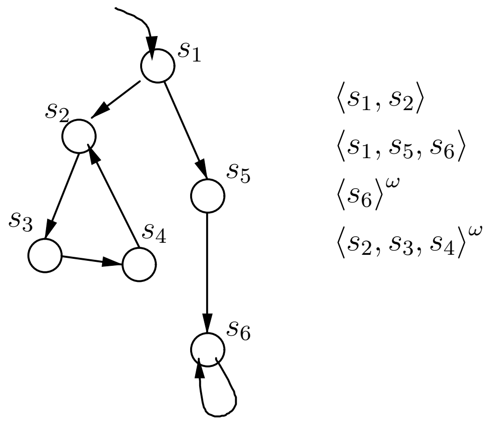



Figure 4.4: Brick representation of a Kripke structure

It is easy to see that the constructed Kripke structure $K_Q$ will be exactly the same as $K$.

**Definition 4.2.3** A set of bricks $Q$ is tree-like if and only if

*(i)* $K_Q$ is tree-like, and

*(ii)* each cycle in $K_Q$ is contained in a brick $l \in Q$.

Condition (ii) ensures that path bricks cannot be combined into loops.

**Example 4.2.3** *In Example 4.2.2, $Q$ is tree-like. However, the following set of bricks are not tree-like since the loop $\langle s_2, s_3, s_4\rangle^\omega$ does not belong to any brick:*

$$\{\langle s_1, s_2, s_3\rangle, \langle s_3, s_4, s_2\rangle, \langle s_1, s_5, s_6\rangle, \langle s_6\rangle^\omega\}$$

**Lemma 4.2.3** Given a tree-like Kripke structure $K$, there exists a tree-like set of bricks $Q$ s.t. $K_Q = K$. We refer to $Q$ as a *construction* of $K$, or say that $Q$ constructs $K$.

For the Kripke structure $K_1$ shown in Figure 4.3, a construction can be

$$\{\langle s_1, s_2\rangle, \langle s_1, s_6, s_7\rangle, \langle s_2, s_3, s_4\rangle^\omega, \langle s_4, s_5\rangle, \langle s_7\rangle^\omega, \langle s_5\rangle^\omega\}$$

Intuitively, a construction is another representation of a tree-like Kripke structure.

### 4.2.1 Indexed Kripke structures

In order to simplify our algorithm, we introduce *indexed* Kripke structures in this subsection.

An indexed Kripke structure $K^n$ is obtained from a Kripke structure $K$ by creating several copies of each state; these copies are distinguished by an index, but cannot be distinguished by temporal formulas. This construction appears to be a formal trick at first sight, but it has the advantage that we can describe traces which lead through the same state of a system several times (but possibly *for different reasons*) without introducing a loop.

Formally, $K^n$ is described as follows:

**Definition 4.2.4** Given a Kripke structure $K = (S, I, R, L)$ and a set of integers $N = \{1, 2, \ldots, n\}$, an *indexed* Kripke structure $K^n = (S^n, I^n, R^n, L^n)$ is defined as follows.

- $S^n = S \times N$, i.e., the states have the form $\langle s, i \rangle$. By convention, we write $s^i$ instead of $\langle s, i \rangle$. $i$ is called the index of the state $s^i$.
- $I^n = I \times N$.
- For any two states $s_1^i, s_2^j \in S^n$, $(s_1^i, s_2^j) \in R^n$ if and only if $(s_1, s_2) \in R$;
- For all states $s^i \in S^n$ we have $L^n(s^i) = L(s)$.

Intuitively, $K^n$ contains $n$ identical copies of each state. It is easy to prove the following lemma.

**Lemma 4.2.4** For all $n$ and $K$, $K$ and $K^n$ are bisimilar, i.e., $K \equiv K^n$.

In particular, for each state $s$, index $i$, and temporal formula $\varphi$ the following holds:

$$K, s \models \varphi \quad \text{iff} \quad K^n, s^i \models \varphi$$

Since our notion of counterexamples is based on the simulation relation, every counterexample over $K^n$ is also a counterexample over $K$. Note that in counterexamples over $K^n$, several copies of the same state in $K$ may appear with different indices.

In the next section, we will discuss an algorithm which generates tree-like counterexamples for **all** $\mathbf{ACTL}$ formulas based on indexed Kripke structures.

Note however that we will not explicitly construct the indexed Kripke structure. Instead, we will just an integer variable to keep track of the index. The indexed Kripke structure model only will serve to make the procedure more transparent.

## 4.3 Generating tree-like counterexamples for ACTL

As defined in Chapter 4.1, a counterexample for a specified property is a Kripke structure $C$ which (i) is simulated by the original Kripke structure, and (ii) disproves the property.

We have argued that it is desirable to generate "simple" counterexamples. We claim that tree-like Kripke structures give rise to such a notion of simple counterexamples. Formally, we say that a counterexample $C$ is tree-like, if $C$ is a tree-like Kripke structure.

The following example shows what tree-like counterexamples look like.

**Example 4.3.1** For an $\mathbf{ACTL}$ formula $\mathbf{AG}\,\neg x \vee \mathbf{AF}\,\neg y$, a tree-like counterexample can look like the structure in Figure 4.5. Furthermore, consider another formula $\mathbf{AF}\ \mathbf{AG}\ \neg x$, whose tree-like counterexample is shown in Figure 4.6.

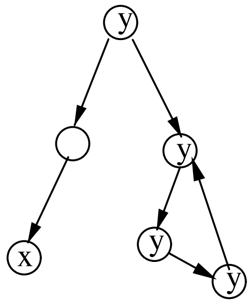



Figure 4.5: Counterexample for $\mathbf{AG}\ \neg x \vee \mathbf{AF}\ \neg y$

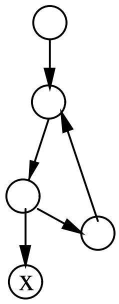



Figure 4.6: Counterexample for $\mathbf{AF}\ \mathbf{AG}\ \neg x$

Tree-like counterexamples have the following advantages:

- Because their structure is similar to trees, they are easy to understand by human users. Moreover, we shall see that subtrees in the tree-like counterexample correspond to counterexamples of subformulas in a natural manner.

- Moreover, the tree structure facilitates simple and effective recursive algorithms. We will demonstrate this in Section 4.4 where we show an abstraction refinement procedure for tree-like counterexamples.

In this section, we discuss an algorithm to generate tree-like counterexamples for all $\mathbf{ACTL}$ formulas. First, we define tree representations of $\mathbf{CTL}$ formulas using their parse trees and sketch how SMV [78] checks $\mathbf{CTL}$ properties.

### 4.3.1 Fixpoint Characterization for $\mathbf{ACTL}$

**Definition 4.3.1** A *parse tree* $Tr_\varphi$ of a $\mathbf{CTL}$ formula $\varphi$ is a tree in which internal nodes are labeled by the operations $\wedge$, $\vee$, $\mathbf{EX}$, $\mathbf{EG}$, $\mathbf{EF}$, $\mathbf{EU}$, $\mathbf{AX}$, $\mathbf{AG}$, $\mathbf{AF}$, and $\mathbf{AU}$. Terminal nodes are labeled by atomic formulas or negated atomic formulas. Note that we assume that negation is only applied to atomic formulas. Therefore, $p \Rightarrow q$ is an abbreviation of $\neg p \vee q$.

As an example, the parse trees for $\mathbf{AG}[p \Rightarrow \mathbf{AF}(q \wedge r)]$ and $\mathbf{EF}[p \wedge \mathbf{EG}(\neg q \vee \neg r)]$ are shown in Figure 4.7. The terminal nodes are labelled by atomic formulas $\neg p$, $p$, $\neg q$, $q$, $\neg r$ and $r$. The internal nodes are labelled by temporal operators $\mathbf{AG}$, $\mathbf{AF}$, $\mathbf{EG}$ and $\mathbf{EF}$ or propositional connectives $\wedge$ and $\vee$. Given a node $v$ in a parse tree, we denote the operator at $v$ by $op(v)$ and the formula sitting at $v$ by $fml(v)$. Furthermore, let $sat(v)$ denote the set of states which satisfy the formula sitting at $v$, i.e., $fml(v)$. Formally, $sat(v) = \{s \mid s \models fml(v)\}$. When the context is clear, we will not distinguish a subformula and its corresponding node in the parse tree.

Given an $\mathbf{ACTL}$ formula $\varphi$, the model checking algorithm in SMV traverses the parse tree corresponding to $\neg\varphi$ in depth-first manner. For example, if $\varphi = \mathbf{AG}[p \Rightarrow \mathbf{AF}(q \wedge r)]$, then SMV works with the parse tree shown in Figure 4.7(b). It is easy to see that the parse tree corresponds to an $\mathbf{ECTL}$ formula.

Note that the parse tree for the negation of an $\mathbf{ACTL}$ formula contains no other temporal operators than $\mathbf{EX}$, $\mathbf{EG}$, $\mathbf{EF}$ and $\mathbf{EU}$. In the case of $\mathbf{EX}$, SMV

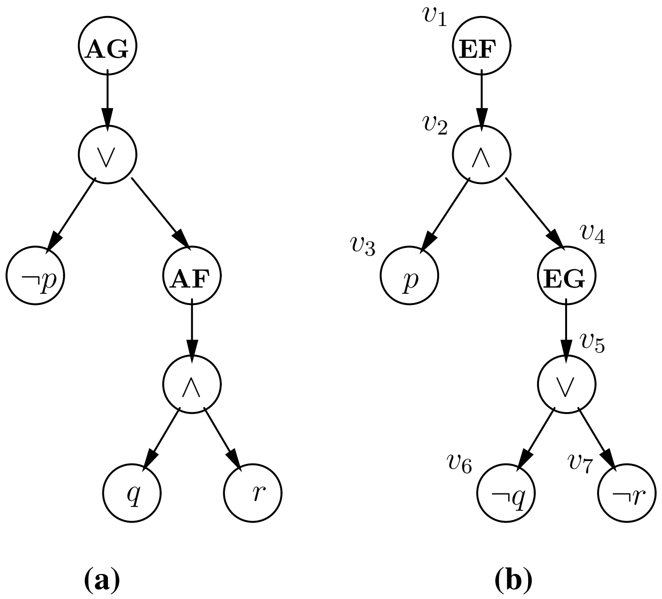



Figure 4.7: The parse tree for $\mathbf{AG}[p \Rightarrow \ \mathbf{AF}(q \wedge r)]$

computes the set of states which satisfy $\mathbf{EX}$ as follows:

- $\mathbf{EX}\, p = \exists s'[TR(s, s') \wedge p(s')]$;

For the other temporal operators, SMV uses fixpoint computation techniques to compute the set of states which satisfies the formula [26, 78]:

- $\mathbf{EF}\, p = \mu Z[p \vee \mathbf{EX}\, Z]$;
- $\mathbf{EG}\, p = \nu Z[p \wedge \mathbf{EX}\, Z]$;
- $\mathbf{E}(p\, \mathbf{U}\, q) = \mu Z[q \vee \mathbf{EX}(p \wedge Z)]$;

In these formulas, $\mu$ is the least fixpoint operator and $\nu$ is the greatest fixpoint operator. The detailed proofs of the fixpoint characterizations can be found in [78].

Given a node $v$, SMV first computes the sets of states which satisfy the subformulas of $fml(v)$. Then it computes the set of states which satisfy $fml(v)$. For example, consider node $v_5$ in Figure 4.7(b). Assume that SMV has already computed $sat(v_6)$ and $sat(v_7)$. Then according to the definition of conjunction $\wedge$ SMV computes $sat(v_5) = sat(v_6) \cap sat(v_7)$. For nodes which are labeled by temporal operators, for example $v_1$ and $v_4$, the model checking algorithm involves the fixpoint computations mentioned above. For example, for the node $v_1$,

$$sat(v_1) = \mu Z[sat(v_2) \cup sat(\mathbf{EX}\, Z)],$$

i.e., $sat(v_1)$ is the least fixpoint of the formula $sat(v_2) \cup sat(\mathbf{EX}\, Z)$.

In general, given a node $v$ with child $u$ where $op(v) = \mathbf{EF}$, $sat(v)$ is computed by $sat(v) = \mu Z\ \tau(Z)$ where $\tau(Z) = sat(u) \cup sat(\mathbf{EX}\, Z)$.

It is well known that the least fixed point $\mu Z.\tau(Z)$ can be computed by iterating the operator $\tau$, starting with the empty set of states, i.e., the set of states satisfying $\texttt{FALSE}$. For a detailed exposition, refer to [26].

More formally, let $\tau^i(\texttt{FALSE}) = \tau(\tau^{i-1}(\texttt{FALSE}))$ for $i > 0$ and $\tau^0(\texttt{FALSE}) = \texttt{FALSE}$. Then $\tau^1(\texttt{FALSE}) = sat(u)$ and $\tau^2(\texttt{FALSE}) = sat(u) \cup sat[\mathbf{EX}\, sat(u)]$. Intuitively, $\tau^i(\texttt{FALSE})$ is the set of states from which a state satisfying $sat(u)$ is reached within $i$ steps. It is easy to see that

$$\tau^0(\texttt{FALSE}) \subseteq \tau^1(\texttt{FALSE}) \subseteq \cdots \subseteq \tau^i(\texttt{FALSE}) \subseteq \cdots.$$

The relation among $\tau^i$ is shown in Figure 4.8.

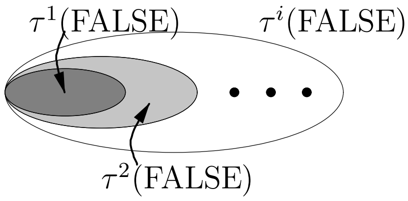



Figure 4.8: Relation among $\tau^i$ in the least fixpoint computation

**Lemma 4.3.1** [26] The sequence $\tau^0(\texttt{FALSE}) \subseteq \tau^1(\texttt{FALSE}) \subseteq \cdots \subseteq \tau^i(\texttt{FALSE}) \subseteq \cdots$ converges to the least fixed point of $\tau$. In other words, $\mu\tau(Z) = \tau^k(\texttt{FALSE})$ for some $k \leq 0$.

The fixed point computation computes all sets $\tau^i(\texttt{FALSE})$ until the sequence converges. Each such set is called a *stage* of the fixpoint computation. By Lemma 4.3.1, the final stage $\tau^k(\texttt{FALSE})$ equals $sat(v)$. We denote the sequence of stages

$$(\tau^1(\texttt{FALSE}), \tau^2(\texttt{FALSE}), \ldots, \tau^k(\texttt{FALSE}))$$

by $stg(fml(v))$.

To compute **EG**, we also need to compute greatest fixpoints. For greatest fixpoints, the fixpoint computation is exactly the same except for one important difference: the iteration starts with the set `TRUE` of all states, and converges to the greatest fixed point from above.

Given an **ECTL** formula $\varphi$, $stg(\varphi) = (S_1, S_2, \ldots, S_n)$ denotes the sequence of sets of states during the fixpoint computation.

In the next subsection, we will describe our counterexample generation algorithm.

### 4.3.2 Algorithms to generate tree-like counterexamples

The algorithm **print_witness** to generate counterexamples for **ACTL** is given as follows.

```
print_witness(v, s_0^m) : {FAIL, SUCCESS}
    if op(v) = EX then
        return print_witnessEX(v, s_0^m)
    if op(v) = EF then
        return print_witnessEF(v, s_0^m)
    if op(v) = EG then
        return print_witnessEG(v, s_0^m)
    if op(v) = EU then
        return print_witnessEU(v, s_0^m)
    if op(v) = ∧ then
        if print_witness(v.Left, s_0^m) = FAIL then
            return FAIL
        return print_witness(v.Right, s_0^m)
    if op(v) = ∨ then
        if print_witness(v.Left, s_0^m) = FAIL then
            return print_witness(v.Right, s_0^m)
        return SUCCESS
    return SUCCESS
```

The procedure **print_witness** takes a parse tree for the negated **ACTL** formula (i.e., an **ECTL** formula) and an initial state from the indexed Kripke structure and returns either `SUCCESS` or `FAIL`. `SUCCESS` denotes that a tree-like counterexample is successfully generated while `FAIL` implies that there is no counterexample. The generated counterexample is output piece by piece in form of a tree-like set of bricks in the subprocedures of **print_witness**.

For a node $v$, $v.Left$ and $v.Right$ means the left and right child of $v$ respectively while $v.Child$ means the only child of node $v$. The procedure is recursive. It uses four other procedures to compute and print counterexamples for **EX**, **EF**, **EU** and **EG** respectively.

The four other procedures use a global integer variable $C$. $C$ is a global variable that is always larger or equal to $m$. The Kripke structure is viewed as an indexed Kripke structure, where $C$ is used as an index. Incrementing $C$ will prevent the procedures from generating the same states in different parts of the counterexample.

The procedure **print_witness EX** is quite simple, and will be omitted here.

The procedures $\mathbf{print\_witness\,EF}$ and $\mathbf{print\_witnessEU}$ are similar. Therefore, we will only explain how $\mathbf{print\_witness\,EF}$ works. At last, we explain the most complicated procedure $\mathbf{print\_witness\,EG}$.

```
print_witnessEF(v, s_0^m)
    (S_1, ..., S_n) = stg(fml(v))
    j = 1
    while (j < n and s_j ∉ S_1)
        S = Img(s_{j-1}) ∩ S_{n-j}
        s_j = pick a state from S
        j = j + 1
    C = C + 1
    print ⟨s_0^m, s_1^C, ..., s_{n-1}^C⟩
    return print_witness(v.Child, s_{n-1}^C)
```

```
print_witnessEU(v, s_0^m)
    (S_1, ..., S_n) = stg(fml(v))
    j = 1
    while (j < n and s_j ∉ S_1)
        S = Img(s_{j-1}) ∩ S_{n-j}
        s_j = pick a state from S
        j = j + 1
    C = C + 1
    print ⟨s_0^m, s_0^C, ..., s_{n-1}^C⟩
    if print_witness(v.Left, s_0^m) = FAIL return FAIL
    for(i=1 to n-2)
        if print_witness(v.Left, s_i^C) = FAIL return FAIL
    return print_witness(v.Right, s_{n-1}^C)
```

Given a parse tree rooted at $v$, assume that $u$ is the direct child of $v$, i.e., $fml(v) = \mathbf{EF}\, fml(u)$. $(S_1, \ldots, S_n) = stg(fml(v))$ is the sequence of the set of states computed by **EF**, i.e., $S_i = \tau^i(\mathtt{FALSE})$. Note that $s_0 \in S_n$. The procedure **print_witnessEF** generates a path $\langle s_0^m, s_1^C, \ldots, s_{n-1}^C \rangle$ where $s_{n-i} \in S_i$. Note that the generated path is labeled by a *new* integer. By labeling this new number, we distinguish these states from the states generated previously. As we have discussed before, $S_1 = sat(u)$, therefore, $s_{n-1} \models sat(u)$. It is easy to see that $\langle s_0, \ldots, s_{n-1} \rangle$ is a witness of $fml(v)$.

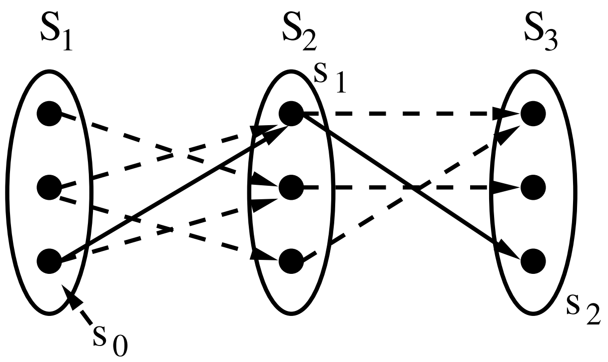



Figure 4.9: Counterexample for a **EF** formula

**Example 4.3.2**

*Consider an example shown in Figure 4.9 where $stg(fml(v)) = (S_1, S_2, S_3)$ and $s_0 \in S_1$, according to the procedure* **print_witnessEF**, *it prints $\langle s_0^m, s_1^C, s_2^C \rangle$ as a counterexample for the indexed Kripke structure.*

Using similar notations, let us consider a node $v$ with $fml(v) = \mathbf{EG}\, fml(u)$. Similarly, let $\tau(Z) = sat(u) \cap sat(\mathbf{EX}\, Z)$, then $sat(v) = \nu Z \tau(Z)$, i.e., $sat(v)$ is the greatest fixpoint of the formula $\tau(Z)$.

As explained above, the greatest fixpoint is obtained by iterating $\tau$ starting from the set of all states, i.e.,

$$\tau^0(\texttt{TRUE}) \supseteq \tau^1(\texttt{TRUE}) \supseteq \cdots \supseteq \tau^i(\texttt{TRUE}) \supseteq \cdots$$

and there exists a $k \geq 0$, such that the greatest fixpoint of $\tau(Z)$ equals $\tau^k(\texttt{TRUE})$.

Then $(S_1, \ldots, S_n) = stg(fml(v))$ records the sequence of sets of states computed by fixpoint algorithm **EG**, i.e., $S_i = \tau^i(\texttt{TRUE})$. Therefore, $S_1 = \tau(\texttt{TRUE}) = sat(u)$ and $S_n = sat(v)$. Since, $S_n = \tau(S_n)$, the following lemma holds.

**Lemma 4.3.2** *Let $M$ be the original Kripke structure and $K$ the Kripke structure restricted to $S_n$, i.e., $K = M \downarrow S_n$. Then $K$ is total.*

```
print_witnessEG(v, s_0^m)
    (S_1, ..., S_n) = stg(fml(v))
    T = {s_0}
    j = 1
    label = 1
    while (T ≠ S)
        S = Img(s_{j-1}) ∩ S_n
        s_j = pick a state from (S − T)      – pick a successor
        Q = Img(s_j) ∩ T
        if (Q ≠ ∅) then break;     – loop discovered
        j = j + 1
        T = T ∪ {s_j}
        s_i = pick a state from Q
    if T = S then return FAIL
    C = C + 1
    print ⟨s_0^m, s_1^C, ..., s_j^C, s_i^C⟩
    print ⟨s_i^C, s_{i+1}^C, ..., s_j^C⟩^ω
    if print_witness(v.Child, s_0^m) = FAIL return FAIL
    for (k = 1 to j)
        if print_witness(v.Child, s_k^C) = FAIL return FAIL
    return SUCCESS
```

According to Lemma 4.3.2, there exists a loop among the states in $S_n$. Our algorithm to find loops in $S_n$ is greedy. $T$ stores all the states which are traversed. Given a state $s_{j-1}$, it discovers a new state $s_j \in S - T$ and checks if we can close the loop. If we can, it stops and returns the loop. If the algorithm cannot close the loop, it continues until $T = S$ which implies that there is no loop.

The procedures **print_witnessEF**, **print_witnessEU** and **print_witnessEG** output bricks, i.e., either K-paths or K-loops. Note that each state in the bricks is marked by an integer. Whenever a brick is generated, the states in the brick are marked by a unique integer which is provided by the global variable $C$. Therefore, two states which appear twice in different procedures are treated as different states because they will have different labels.

In order to understand how the algorithm **print_witness** works, let us con-

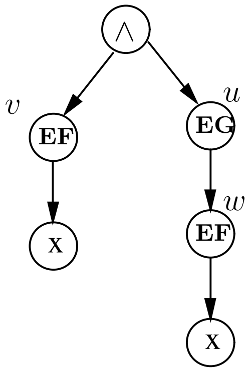



Figure 4.10: The parse tree for $\mathbf{EF}\, x \wedge \mathbf{EG}\,\mathbf{EF}\, x$

sider an $\mathbf{ACTL}$ property.

**Example 4.3.3** Assume that an $\mathbf{ACTL}$ property $\varphi = \mathbf{AG}\,\neg x \vee \mathbf{AF}\,\mathbf{AG}\,\neg x$. Then the parse tree for $\neg\varphi$ is shown in Figure 4.10. A set of bricks $Q$ generated by **print_witness** may be

$$
\begin{array}{lll}
\langle s_0^0, s_1^1, s_2^2\rangle & -- & \text{counterexample for } \mathbf{AG}\,\neg x \\
\langle s_0^0, s_3^3, s_0^4\rangle & -- & \text{counterexample for } \mathbf{AF}\,\mathbf{AG}\,\neg x \\
\langle s_0^4, s_3^5\rangle^\omega & -- & \text{counterexample for } \mathbf{AF}\,\mathbf{AG}\,\neg x \\
\langle s_0^4, s_2^7\rangle & -- & \text{counterexample for } \mathbf{AG}\,\neg x \\
\langle s_3^5, s_2^8\rangle & -- & \text{counterexample for } \mathbf{AG}\,\neg x.
\end{array}
$$

The corresponding constructed counterexample $K_Q$ is shown in Figure 4.11. It is easy to see that $T$ is a tree-like Kripke structure. Note that $s_0, s_2, s_3$ are repeated several times in the counterexample. However, they are treated as different states since they have different marks.

It is easy to see that the program **print_witness** terminates if the $\mathbf{ACTL}$ formula is finite.

**Lemma 4.3.3** The program **print_witness** terminates.

Therefore, the maximal value for the global variable $C$ is finite. Let us assume that this value is $c$. Then the following theorem holds.

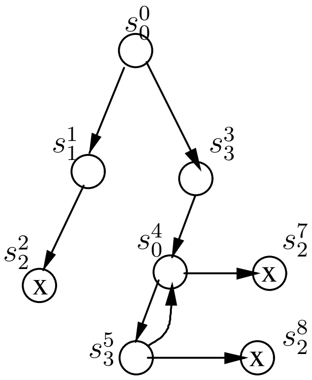



Figure 4.11: Counterexample for $\mathbf{AG}\,\neg x \vee \mathbf{AF}\,\mathbf{AG}\,\neg x$

**Theorem 4.3.1** Given an $\mathbf{ACTL}$ formula $\varphi$, a Kripke structure $M = (S, \{s_0\}, R, L)$ such that $M \not\models \varphi$, let $Q$ be the set of bricks generated by $\mathbf{print\_witness}(\neg\varphi, s_0)$. Then $K_Q$ is a tree-like counterexample for $\varphi$ on $M^c$.

This theorem guarantees that our algorithm will generate a tree-like counterexample for the indexed Kripke structures.

## 4.4 Refinement algorithm for $\mathbf{ACTL}$

In the previous section, we showed that there are tree-like counterexamples for all $\mathbf{ACTL}$ formulas. In particular, we provided an algorithm to generate tree-like counterexamples for such formulas. The algorithm can be potentially extended to handle $\mathbf{ACTL}^\star$ as well. In this section, we extend our counterexample-guided abstraction refinement methodology (see Chapter 3) for all the formulas in $\mathbf{ACTL}$.

Recall that our counterexample-guided abstraction refinement methodology works as follows. Given a Kripke structure $M$ and an $\mathbf{ACTL}$ property $\varphi$, we first generate initial abstraction functions and build the initial abstract Kripke structure $\widehat{M}$ accordingly. Then the traditional model checker will check if $\varphi$ holds on $\widehat{M}$. If not, it will generate a counterexample. The next step is to check if the counterexample is spurious or not.

As we discussed in the previous section, **print_witness** will generate a tree-like set of abstract bricks $\widehat{Q}$ from which an abstract counterexample $K_{\widehat{Q}}$ can be derived. The algorithm shown in Figure 4.12 is a recursive procedure based on depth-first traversal of the abstract counterexample. It takes an abstract initial state $\widehat{s_0}^m$ marked by $m$ and the set of bricks $\widehat{Q}$ as arguments. Initially, we call **CheckRefine**($\widehat{s_0}^0$, $\widehat{Q}$). Note that $\widehat{s_0}^0$ is the initial state of $K_{\widehat{Q}}$. **CheckRefine** returns the set of states in $h^{-1}(\widehat{s_0}^m)$ which have concrete tree-like counterexamples. If this set is empty, that implies that the abstract counterexample is spurious.

**CheckRefine**($\widehat{s_0}^m$, $\widehat{Q}$)
  $T = h^{-1}(\widehat{s_0})$
  **foreach** $\widehat{q} \in \widehat{Q}$
    **if** $\widehat{q}_1 \neq \widehat{s_0}^m$ **continue**
    $len = |\widehat{q}|$
    $S_1 = h^{-1}(\widehat{s_0})$
    **for** $(i = 2 \textbf{ to } len)$
      $S_i$ = CheckRefine($\widehat{q}_i$, $\widehat{Q}$)
    **if** $\widehat{q}$ is a path **then**
      $T = T \cap \textbf{CheckPATH}(S_1, S_2, \ldots, S_{len})$
    **if** $\widehat{q}$ is a loop **then**
      $T = T \cap \textbf{CheckLOOP}(S_1, S_2, \ldots, S_{len})$
  return $T$

Figure 4.12: Refinement algorithm for all **ACTL** formulas

In the procedure **CheckRefine**, $T$ denotes the set of concrete states which are the initial states of some concrete counterexamples. Note that for any state $s \in T$, $h(s) = \widehat{s_0}$. For a brick $q = \langle s_1, \ldots, s_n \rangle$, $q_i$ denotes the i-th element of $q$, i.e., $s_i$. Given a set of bricks $Q$, **CheckRefine** first checks all the bricks which start from $\widehat{s_0}^m$. For such a brick $\widehat{p}$, the procedure recursively checks the bricks that start from some states in $\widehat{p}$. If $anchor(\widehat{q}) \in \widehat{p}$, then the

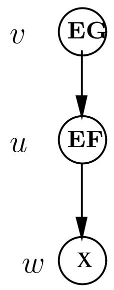
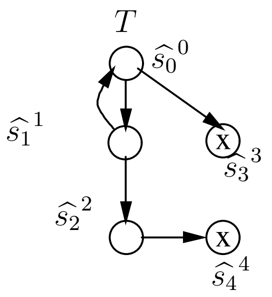



Figure 4.13: The parse tree for **AF AG** $\neg x$

procedure checks $\widehat{q}$ first and returns the set of states from which the concrete counterexample corresponding to $\widehat{q}$ can be constructed. Using the obtained sets of states, **CheckRefine** checks whether the current abstract trace corresponds to a concrete trace by using two subroutines **CheckPATH** and **CheckLOOP**. **TO DO:** example

The function **CheckLOOP** checks if a sequence of sets of states $\langle S_0, \ldots, S_j \rangle$ includes concrete loops or not. If not, it will use **PolyRefine** to refine the abstraction. Otherwise, it returns a set of initial states $S \subset S_0$ which will lead to a loop. The function **CheckPATH** is similar. These two functions are closely related to the algorithm **SplitLOOP** and **SplitPATH** in Chapter 3.3. The difference is that both **CheckLOOP** and **CheckPATH** need to return the set of concrete initial states.

**Example 4.4.1** Consider an ACTL property $\varphi$ = **AF AG** $\neg x$. The parse tree for $\neg\varphi$ is in Figure 4.13. Assume that $\varphi$ does not hold on the abstract Kripke structure $\widehat{M}$. Also assume that the corresponding counterexample $T$ for $\varphi$ is shown in Figure 4.14. The algorithm **CheckRefine** considers the trace $\langle \widehat{s_0}^0, \widehat{s_3}^3 \rangle$ first. It uses **CheckPATH** to determine whether there exists a concrete path from a state in $h^{-1}(\widehat{s_0})$ to a state in $h^{-1}(\widehat{s_3})$. If not, then $\langle \widehat{s_0}^0, \widehat{s_3}^3 \rangle$ is spurious. Then **PolyRefine** is used to refine the abstraction. If there exists a concrete path, **CheckPATH** will return the set of states in $h^{-1}(\widehat{s_0})$ from which




$Q$ includes

$\langle \widehat{s_0}^0, \widehat{s_3}^3 \rangle$

$\langle \widehat{s_0}^0, \widehat{s_1}^1 \rangle^\omega$

$\langle \widehat{s_1}^1, \widehat{s_2}^2, \widehat{s_4}^4 \rangle$

Figure 4.14: Counterexample for **AF AG** $\neg x$

the concrete paths start, i.e.,

$$S_0 = \{s \mid \exists \pi, \pi^0 = s, \pi^1 \in h^{-1}(\widehat{s_3})\}.$$

A similar process is applied to $\langle \widehat{s_1}^1, \widehat{s_2}^2, \widehat{s_4}^4 \rangle$. If there exists a concrete path associated with this abstract path, it will return the set of states $S_1 \subseteq h^{-1}(\widehat{s_1}^1)$ from which the concrete paths start.

Next, **CheckRefine** considers the trace $\langle \widehat{s_0}^0, \widehat{s_1}^1 \rangle^\omega$. The function **Check-LOOP** checks whether there exists a loop in $S_0 \cup S_1$. If not, $\langle \widehat{s_0}^0, \widehat{s_1}^1 \rangle^\omega$ is a spurious loop counterexample. We can still use **PolyRefine** to refine the abstraction function.

The algorithm provided in this section can be adopted to refining abstraction for $\mathbf{ACTL}^\star$ formulas.

# Chapter 5

# Abstract BDDs

In this chapter, we describe a data structure - abstract BDDs (aBDDs) which facilitates the abstraction operation. Abstract BDDs (aBDDs) are obtained from ordinary BDDs by merging BDD nodes whose abstract values coincide. We discuss four types of abstract BDDs (called S-type, 0-type, 1-type and $\vee$-type aBDDs) which have found applications in many CAD-related areas such as equivalence checking, variable ordering and model checking.

In the following three sections, we first discuss how a single abstraction function $h : D \rightarrow A$ where $D = B^k$ is applied to a Boolean function $f : D \rightarrow B$. Then we show how to extend the definition to the case with multiple abstraction functions. In the last section, we summarize different types of aBDDs.

## 5.1 Abstract Binary Decision Trees

An abstract BDD (aBDD) is obtained by collapsing all paths corresponding to the same equivalence class into a single path. While such a collapse leads to a loss of information, the size of the aBDD may be significantly decreased, and certain problems may become feasible.

The above mentioned collapsing operation can be defined in various ways, and thus, different types of aBDDs are obtained. Since the concepts underlying aBDDs are most easily explained using Binary Decision *Trees*, we will usually outline this special case first, and show later how to deal with the general case.

As described in Chapter 2.3, the abstraction function $h$ induces an auto-abstraction function $\mathcal{H} : D \to D$. Given a Boolean function $f : D \to B$ and a BDT $T_f$, let $u, v$ be two leaves of $T_f$ (note that $\vec{u}, \vec{v} \in D$) such that $\vec{u} \equiv_h \vec{v}$ and $\mathcal{H}(\vec{v}) = \vec{u}$, i.e., $\vec{u}$ and $\vec{v}$ are equivalent with respect to $\equiv_h$, and $\vec{u}$ is the representative of their equivalence class. In this case we say that the node $u$ is the representative of $v$.

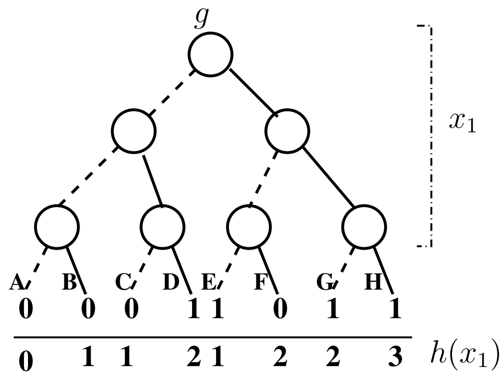



Figure 5.1: The BDT for $g$

**Example 5.1.1** Assume that for $D = B^3$ a Boolean function $g : D \to B$ is given by

$$x_1 = \langle 011 \rangle \vee x_1 = \langle 100 \rangle \vee x_1 = \langle 110 \rangle \vee x_1 = \langle 111 \rangle.$$

The BDT for $g$ is depicted in Figure 5.1. We consider the abstraction function $h(x_1) = count1(x_1)$ which counts the number of 1s in $x_1$. For example, $count1(\langle 101 \rangle) = 2$. The abstract values for each node are listed in Figure 5.1. It is easy to see that $h$ induces an equivalence relation $\equiv_h$ on 0-1 vectors of length 3. For example, $\vec{B} \equiv_h \vec{C} \equiv_h \vec{E}$ since they have the same abstract value 1. Assume that the representative of an equivalence class is the lexicographically least vector in that equivalence class. Then $\vec{B}$ is the representative for the equivalence class with respect to the abstract value 1, i.e.,

$$\mathcal{H}(\vec{B}) = \mathcal{H}(\vec{C}) = \mathcal{H}(\vec{E}) = \vec{B}.$$

The representative nodes for the abstract values $0, 1, 2, 3$ are $A, B, D$, and $H$ respectively. Consider the representative node $B$ which represents $B, C$, and $E$. In this case, we need to collapse the paths leading to $B, C$, and $E$ into a single path leading to $B$.

Different operations can be defined for representative nodes and non-representative nodes. This will result in different types of abstract BDDs. In the following sections, we will define four types of aBDDs for the case of a single variable $x_1$. Later, we will generalize the definitions to the full case.

## 5.2 S-type Abstract BDDs

Given a Boolean function $f : D \rightarrow B$ and a BDT $T_f$, the S-type aBDT $\mathcal{H}^s(f)$ of $T_f$ rooted at $v$ is defined by

$$\mathcal{H}^s(f)(\vec{v}) = f(\mathcal{H}(\vec{v})).$$

To understand this definition just note that $f(\mathcal{H}(\vec{v}))$ is computed as follows: first, the input vector $\vec{v}$ is transformed into its representative $\mathcal{H}(\vec{v})$, and then, $f$ is applied to the representative. In other words, the output of $f$ for the representative determines the output of $f$ for all the other members of the equivalence class.

**Example 5.2.1** *For the Boolean function in Example 5.1.1, the S-type aBDT is shown in Figure 5.2(a). $E$ is a non-representative node. The value at $E$ is overwritten by the value at $B$ which is a representative of $E$. A similar situation occurs at nodes $C, F, G$. The final reduced BDD is in Figure 5.2(b).*

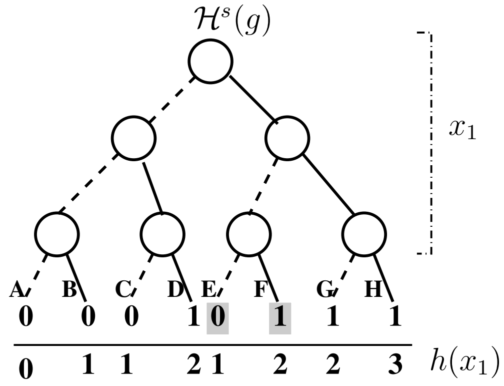



Figure 5.2: S-type aBDT for $g$

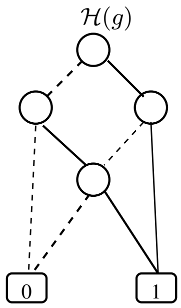



Figure 5.3: S-type aBDD for $g$

*Intuitively, the construction maintains some "useful"* minterms *and ignores other "uninteresting"* minterms.

S-type aBDTs distribute over any logic operations. The following lemma holds.

**Lemma 5.2.1** *Let $f, p, q : D \to B$ be three Boolean functions, $\circ$ be composition and $\odot$ be any logical operation. Then if $f = p \odot q$, then $\mathcal{H}^s(f) = \mathcal{H}^s(p) \odot \mathcal{H}^s(q)$. If $f = p \circ q$, then $\mathcal{H}^s(f) = p \circ \mathcal{H}^s(q)$.*

**Proof** Let $\vec{v} \in D$ be an arbitrary vector. For the first case, we have the following equations:

$$\begin{aligned}\mathcal{H}^s(f)(\vec{v}) &= f(\mathcal{H}(\vec{v})) \\ &= p(\mathcal{H}(\vec{v})) \odot q(\mathcal{H}(\vec{v})) \\ &= \mathcal{H}^s(p)(\vec{v}) \odot \mathcal{H}^s(q)(\vec{v})\end{aligned}$$

Then, for the second case, we have

$$\begin{aligned}\mathcal{H}^s(f)(\vec{v}) &= f(\mathcal{H}(\vec{v})) \\ &= p(q(\mathcal{H}(\vec{v}))) \\ &= p(\mathcal{H}^s(q)(\vec{v}))\end{aligned}$$

□

Assume that $u, t$ are two nodes on level $i+1$ ($i < k$) in the BDT $T_f$. As before, $\vec{u}, \vec{t}$ denote the labels concatenated along the paths from the root to $u$ and $t$ respectively. The concatenation $\vec{u} \cdot e$ where $e \in B^{k-i}$ describes a path to a leaf in $T_f$ because $\vec{u} \cdot e \in D$. We define an equivalence relation $\equiv_i$ over $B^i$ (i.e., over 0-1vectors of length $i$) by

$$\vec{u} \equiv_i \vec{t} \text{ iff } \forall e \in B^{k-i}. h(\vec{u} \cdot e) = h(\vec{t} \cdot e)$$

```
function AbsSame(v)
i = level(v) − 1;
w⃗ = H_i(v⃗);
if w⃗ ≠ v⃗
    return w;
else
    if nonterminal(v)
        left(v) = AbsSame(left(v));
        right(v) = AbsSame(right(v));
    endif;
    return v;
endif
```

Figure 5.4: S-type Abstraction for BDTs

It is easy to see that $\equiv_i$ is reflexive, symmetric and transitive. Similar to the definition of the auto-abstraction function $\mathcal{H}$, $\mathcal{H}_i(\vec{u})$ is defined to select a unique representative from the equivalence class $[\vec{u}]_{\equiv_i}$. Formally, let $rep_i : [B^i]_{\equiv_i} \rightarrow B^i$ be a function which selects a unique representative from $[B^i]_{\equiv_i}$. Then $\mathcal{H}_i(\vec{u}) : B^i \rightarrow B^i$ is defined as

$$\mathcal{H}_i(\vec{u}) = rep_i([\vec{u}]_{\equiv_i}).$$

Note that $\equiv_k$ coincides with $\equiv_h$, i.e., we can view the equivalence relations $\equiv_1, \equiv_2, \ldots$ as approximations of $\equiv_h$.

Next, we show how to compute an S-type aBDT from a given BDT. Without loss of generality, assume that each equivalence class is represented by its lexicographically minimal element. Therefore, if $\leq$ denotes lexicographical ordering, and $\mathcal{H}(\vec{w}) = \vec{w}$, then $\vec{w}$ is the lexicographically first vector in $\{\vec{v} : \vec{v} \equiv_h \vec{w}\}$.

The algorithm **AbsSame** of Figure 5.4 constructs an S-type aBDT is given in. Its argument is a node $v$ in the given BDT. The initial call to the algorithm is **AbsSame**($root_f$) where $root_f$ denotes the root node of $T_f$. In **AbsSame**, the function $level(v)$ returns the level of the node $v$. If $v$ is a representative node, the program will recursively call **AbsSame** to build the subtree of the function. If $v$ is not a representative node, i.e., $\vec{w} \neq \vec{v}$, the program will return the representative node $w$. In other words, $v$ is replaced by $w$.

**Theorem 5.2.1** *Given an abstraction function $h$, if the representative is determined by the lexicographically least vector, the algorithm* **AbsSame** *correctly builds the S-type aBDT for a Boolean function $f$, i.e.,* $\mathbf{AbsSame}(root_f) = \mathcal{H}^s(f)$.

**Proof** Given a path $p$, let $n(p)$ denote the corresponding node where $p$ ends in the original BDD and $n_{\mathcal{H}}(p)$ be the corresponding node where $n(p)$ maps to in $\mathbf{AbsSame}(root_f)$. Note that $n(\vec{v}) = v$. In order to prove the theorem, it is sufficient to show that $n(\mathcal{H}(p)) = n_{\mathcal{H}}(p)$ for all the paths $p$.

We prove this theorem by induction on the length of path $p$. First, the root is considered when $p = \epsilon$, i.e., $n(p) = root$, it is trivial to see that $n_{\mathcal{H}}(p) = n(\mathcal{H}(p) = root$. Let us assume that $n_{\mathcal{H}}(p) = n(\mathcal{H}(p)$ is true for $|p| = i$. Then consider a path $p \cdot y$ where $y \in B$. It is easy to see that both $n_{\mathcal{H}}(p \cdot y)$ and $n(\mathcal{H}(p \cdot y))$ are at level $i + 1$. There are two cases:

- $p \cdot y = \mathcal{H}(p \cdot y)$, from program in Figure 5.4, we know that

$$n_{\mathcal{H}}(p \cdot y) = n(\mathcal{H}(p \cdot y))$$

- $p \cdot y \neq h(p \cdot y)$, from induction hypothesis, we have

$$n_{\mathcal{H}}(p) = n(\mathcal{H}(p))$$

Furthermore according to program in Figure 5.4,

$$n_{\mathcal{H}}(p \cdot y) = n_(\mathcal{H}(\mathcal{H}(p) \cdot y))$$

Since $\mathcal{H}$ is idempotent, $\mathcal{H}(p) = \mathcal{H}(\mathcal{H}(p))$ implies $\mathcal{H}(p \cdot y) = \mathcal{H}(\mathcal{H}(p) \cdot y)$ according to the definition of consistent function. Thus, we have

$$n_{\mathcal{H}}(p \cdot y) = n_(\mathcal{H}(p \cdot y)).$$

By induction, this is true for the last level, which implies $\mathbf{AbsSame}(root_f) = \mathcal{H}^s(f)$. $\Box$

Note that LBDDs are obtained from BDTs by merging isomorphic subtrees. In order to build S-type LBDDs directly from LBDDs, we must modify the algorithm **AbsSame**. The new algorithm **AbsSameM** is described in Figure 5.5. In the algorithm, $Sub(v)$ denotes the subgraph rooted at the node $v$. $Sub(v) \approx Sub(v')$ means that the respective subgraphs rooted at $v$ and $v'$ are isomorphic. The initial call of the algorithm is $\mathbf{AbsSameM}(root_f, \langle\rangle)$.

```
function AbsSameM(v, path)
i = level(v) − 1;
w⃗ = H_i(path);
if w⃗ ≠ path
    return w;
else
    if nonterminal(v)
        left(v) = AbsSameM(left(v), path · 0);
        right(v) = AbsSameM(right(v), path · 1);
        if there exists v_1 in cache such that Sub(v) ≈ Sub(v_1)
            return v_1;
        endif;
    endif;
    return v;
endif
```

Figure 5.5: Modified **AbsSameM** for LBDDs

**Lemma 5.2.2** *For a given LBDD $f$, the size of $\mathcal{H}^s(f)$ is less than or equal to $2 + \prod_{i=1}^{n} |[B^i]_{\equiv_i}|$.*

**Proof** Let $\vec{u}$ and $\vec{t}$ be two paths of length $i$ such that $\vec{u} \equiv_i \vec{t}$. According to our algorithm, $u = t$. Thus, if two paths belong to the same equivalence class, then they lead to the same node. Hence, the number of nodes in the LBDD of $\mathcal{H}^s(f)$ is bounded by the number of equivalence classes defined by $\equiv_i$ at each level. There are $n$ levels of internal nodes and two terminal nodes. Therefore, the size of LBDD $\mathcal{H}^s(f)$ is bounded by $2 + \prod_{i=1}^{n} \mid \equiv_i \mid$. $\Box$

Given an LBDD $f$, the BDD $\mathcal{H}^s(f)$ obtained from algorithm **AbsSameM** is called an abstract LBDD. If we apply the BDD reduction rules on an abstract LBDD, we will obtain an abstract BDD or an aBDD.

In practice, when $|A| \ll |D|$, i.e., the range of the abstraction function $h$ is much smaller than the domain of $h$, the abstraction overhead and the resulting S-type aBDD size are usually very small. As a matter of fact, for many interesting abstraction functions (e.g. modulus, count1, logarithm, linear, partition functions), the abstraction overhead is polynomial in $|A|$.

**Boolean functions of several vector variables.** Given a Boolean function $f : D^n \to B$, the S-type aBDD $\mathcal{H}^s(f)$ of $f$ is defined by

$$\mathcal{H}^s(f)(x_1, \ldots, x_n) = f(\mathcal{H}(x_1), \ldots, \mathcal{H}(x_n)).$$

The properties and algorithms given for a single abstraction can be easily generalized for the multiple abstractions case.

**Lemma 5.2.3** *Let $f, p, q : D^n \to B$ be three Boolean functions, $\circ$ be composition and $\odot$ be any logical operation. Then we have*

$$\begin{array}{ll} f = p \odot q & \to \mathcal{H}^s(f) = \mathcal{H}^s(p) \odot \mathcal{H}^s(q) \\ f = p \circ q & \to \mathcal{H}^s(f) = p \circ \mathcal{H}^s(q) \end{array}$$

The algorithm for generating aBDDs for multiple abstraction functions are shown in Figure 5.6.

```
function AbsSameM(v, p)
i = level(v) − 1;
for (j=1 to n)
    p_j = cut(p, x_j);
    q_j = H_i(p_j);
w⃗ = q_1 · q_2 · ··· · q_n;
if w⃗ ≠ p
    return w;
else
    if nonterminal(v)
        left(v) = AbsSameM(left(v), p · 0);
        right(v) = AbsSameM(right(v), p · 1);
        if there exists v_1 in cache such that Sub(v) ≈ Sub(v_1)
            return v_1;
        endif;
    endif;
    return v;
endif
```

Figure 5.6: **AbsSameMult** for multiple abstraction functions

## 5.3 0-type And 1-type Abstract BDDs

In this section, we introduce two new types of aBDDs (0-type and 1-type aBDDs) which have different properties from S-type BDDs. In particular we shall see that there is a clear relation between 0(1)-type aBDDs and the original functions.

Recall that in the abstraction procedure for S-type aBDDs, non-representative nodes are replaced by representative nodes. In 0(1)-type aBDDs, however, non-representative nodes are replaced by 0(1)-nodes, i.e., the representative nodes remain unchanged, and all others are uniformly set to output either 0 or 1. Formally, the 0(1)-type abstract BDD $\mathcal{H}^0(f)$(or $\mathcal{H}^1(f)$) of $T_f$ rooted at $v$ is defined as

$$\mathcal{H}^0(f)(\vec{v}) = (\mathcal{H}(\vec{v}) \leftrightarrow \vec{v}) \wedge f(\vec{v}) = \begin{cases} f(\vec{v}) & \text{if } \mathcal{H}(\vec{v}) = \vec{v} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{H}^1(f)(\vec{v}) = (\mathcal{H}(\vec{v}) \leftrightarrow \vec{v}) \rightarrow f(\vec{v}) = \begin{cases} f(\vec{v}) & \text{if } \mathcal{H}(\vec{v}) = \vec{v} \\ 1 & \text{otherwise} \end{cases}$$

**Example 5.3.1** *Consider again the Boolean function defined in Example 5.1.1. The 0-type and 1-type aBDD for $g$ are shown in Figure 5.7 and Figure 5.8.*

0(1)-type aBDDs do not directly distribute over logic operations. However, the following lemma holds.

**Lemma 5.3.1** *Let $f, p, q : D \rightarrow B$ be three Boolean functions, $\circ$ is composition and $\odot$ be any logical operation. Then we have*

$$\begin{array}{lll} f = p \odot q & \rightarrow & \mathcal{H}^0(f) = \mathcal{H}^0(\mathcal{H}^0(p) \odot \mathcal{H}^0(q)) \\ f = p \odot q & \rightarrow & \mathcal{H}^1(f) = \mathcal{H}^1(\mathcal{H}^1(p) \odot \mathcal{H}^1(q)) \\ f = p \circ q & \rightarrow & \mathcal{H}^0(f) = \mathcal{H}^0(p) \circ \mathcal{H}^0(q) \\ f = p \circ q & \rightarrow & \mathcal{H}^1(f) = \mathcal{H}^1(p) \circ \mathcal{H}^1(q) \end{array}$$

**Proof** Let $\vec{v} \in D$ be an arbitrary vector. In the following, we prove the case for 0-type. The case of 1-type follows the same proof. When $f = p \odot q$, if $\vec{v}$ is a representative,

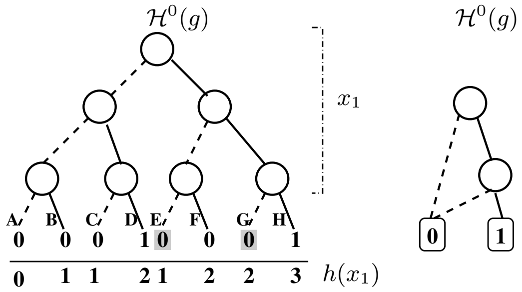
$$\begin{aligned} \mathcal{H}^0(f)(\vec{v}) &= f(\mathcal{H}(\vec{v})) \\ &= p(\mathcal{H}(\vec{v})) \odot q(\mathcal{H}(\vec{v})) \\ &= \mathcal{H}^0(p)(\vec{v}) \odot \mathcal{H}^0(q)(\vec{v}) \end{aligned}$$

If $\vec{v}$ is not a representative, $\mathcal{H}^0(f)(\vec{v}) = 0$. Let $g = \mathcal{H}^0(p) \odot \mathcal{H}^0(q)$. $\mathcal{H}^0(g)(\vec{v}) = 0$ for 0-type as well. Therefore, $\mathcal{H}^0(f)(\vec{v}) = \mathcal{H}^0(g)(\vec{v})$.

When $f = p \circ q$, if $\vec{v}$ is a representative,

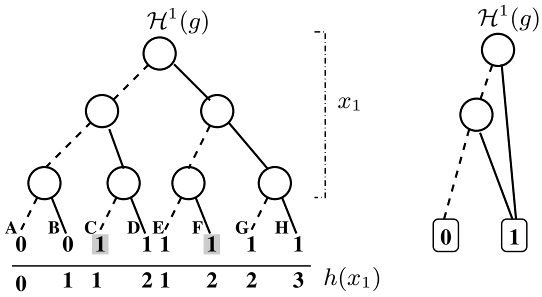
$$\begin{aligned} \mathcal{H}^0(p) \circ \mathcal{H}^0(q)(\vec{v}) &= p(\mathcal{H}^0(q)(\mathcal{H}(\vec{v}))) \\ &= p(q(\mathcal{H}(\mathcal{H}(\vec{v})))) \\ &= p(q(\mathcal{H}(\vec{v}))) \\ &= f(\mathcal{H}(\vec{v})) = \mathcal{H}^0(f)(\vec{v}) \end{aligned}$$

If $\vec{v}$ is not a representative, $\mathcal{H}^0(f)(\vec{v}) = 0 = \mathcal{H}^0(p) \circ \mathcal{H}^0(q)(\vec{v})$. Overall, the statements hold for all vectors $\vec{v} \in D$. $\square$




(a) 0-type aBDT and aBDD

Figure 5.7: 0-type aBDD for $g$




(b) 1-type aBDT and aBDD

Figure 5.8: 1-type aBDD for $g$

The algorithm **AbsZero** of Figure 5.9 is a modification of the algorithm $AbsSameM$ for 0-type abstract LBDDs. A similar algorithm **AbsOne** can be used to build 1-type abstract LBDDs. The only difference between **AbsZero** and **AbsOne** is that **AbsOne** returns 1 and **AbsZero** returns 0 when $path$ is not a representative.

Theorem 5.2.1 and Lemma 5.2.2 analogously hold for 0(1) type abstract LBDDs. Furthermore, the following lemma shows that the 0(1) type aBDDs can be viewed as lower (respectively upper) approximations of the original function.

```
function AbsZero(v, path)
i = level(v);
w⃗ = H_i(path);
if w⃗ ≠ path
    return 0;
else
    if nonterminal(v)
        left(v) = AbsZero(left(v), path · 0);
        right(v) = AbsZero(right(v), path · 1);
        if there exists v_1 in cache such that Sub(v) ≈ Sub(v_1)
            return v_1;
        endif;
    endif;
    return v;
endif
```

Figure 5.9: Modified **AbsZero** for LBDDs

Following the same argument as Theorem 5.2.1, the algorithms **AbsZero** and **AbsOne** correctly compute the 0(1)-type abstract LBDDs, i.e., **AbsZero**($root_f$)=$\mathcal{H}^0(f)$ and **AbsOne**($root_f$)=$\mathcal{H}^1(f)$. At the same time, Lemma 5.2.2 also holds for 0(1)-type abstract LBDDs.

**Lemma 5.3.2** *The following tautologies hold.*

$$\mathcal{H}^0(f) \to f \quad \text{and} \quad f \to \mathcal{H}^1(f).$$

The lemma can be easily proved. Therefore, the proof is omitted.

Similar as for S-type aBDDs, the approach can be extended to several vector variables and abstraction functions. For a function $f : D^n \to B$, we define

$$\mathcal{H}^0(f)(x_1, \dots, x_n) = \begin{cases} f(x_1, \dots, x_n) & \mathcal{H}(x_1) = x_1, \dots, \mathcal{H}(x_n) = x_n \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{H}^1(f)(x_1, \dots, x_n) = \begin{cases} f(x_1, \dots, x_n) & \mathcal{H}(x_1) = x_1, \dots, \mathcal{H}(x_n) = x_n \\ 1 & \text{otherwise} \end{cases}$$

The properties and algorithms can be easily extended to this general case.

## 5.4 $\vee$-type Abstract BDDs

S-type, 0-type and 1-type aBDDs are similar in the sense that the representative nodes remain unchanged while the non-representative nodes are replaced by new functions.

In this section, we define $\vee$-type aBDDs. For $\vee$-type aBDDs, both representative nodes and non-representative nodes are modified. Recall that in a BDT $T_f$, $\vec{v}$ is the path from the root to node $v$ in a BDT $T_f$. The $\vee$-type aBDT $\mathcal{H}^{\vee}(f)$ corresponding to the Boolean function $f : D \to B$ is defined by the following equation:

$$\mathcal{H}^{\vee}(f)(\vec{v}) \;=\; \begin{cases} \bigvee_{\vec{u} \equiv_h \vec{v}} f(\vec{u}) & \text{if } \vec{v} = \mathcal{H}(\vec{v}) \\ 0 & \text{otherwise} \end{cases}$$

Hence, in the obtained aBDT $\mathcal{H}^{\vee}(f)$, the Boolean function of a representative node is the *disjunction* of all the Boolean functions corresponding to the nodes in the same equivalence class, i.e., the representative node outputs 1 if at least one of its equivalent nodes outputs 1. For non-representative nodes, the Boolean function is defined to be *false* or $0$, similar as in 0-type aBDTs.

**Example 5.4.1** *To illustrate how to build $\vee$-type aBDDs, let us return to Example 5.1.1. According to our definition,*

$$\mathcal{H}_g^{\vee}(\vec{B}) = g(\vec{B}) \vee g(\vec{C}) \vee g(\vec{E}) = 1.$$

*The $\vee$-type aBDT for $g$ is shown in Figure 5.10.*

The next lemma describes how the auto-abstraction function $\mathcal{H}$ interacts with conjunctions and disjunctions.

**Lemma 5.4.1** *Let $f, p, q : D \to B$ be Boolean functions. Then the following tautologies hold:*

$$(f = p \vee q) \;\;\to\;\; (\mathcal{H}^{\vee}(f) = \mathcal{H}^{\vee}(p) \vee \mathcal{H}^{\vee}(q))$$

$$(f = p \wedge q) \;\;\to\;\; (\mathcal{H}^{\vee}(f) \to \mathcal{H}^{\vee}(p) \wedge \mathcal{H}^{\vee}(q))$$

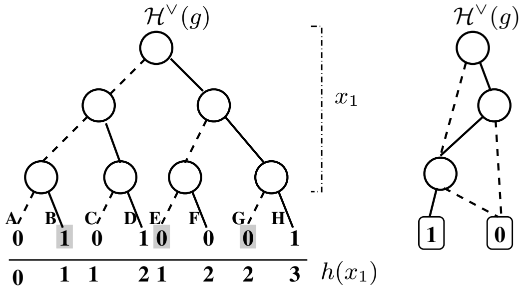



Figure 5.10: $\vee$-type aBDD for $g$

**Proof** Assume that $\vec{v}$ is a $k$-bit 0-1 vector. If $\vec{v}$ is a representative, according to the definition, $\mathcal{H}^{\vee}(f)(\vec{v}) = \bigvee_{\mathcal{H}^{\vee}(\vec{v'})=\vec{v}} f(\vec{v'})$; Otherwise, $\mathcal{H}^{\vee}(f)(\vec{v}) = 0$. The same formula holds when $f$ is replaced by $p$ and by $q$. In the following, we will prove the case where $f = p \wedge q$. The case of $f = p \vee q$ can be easily proved following the same proof.

When $\vec{v}$ is a non-representative, it is easy to see that $\mathcal{H}^{\vee}(p)(\vec{v}) \wedge \mathcal{H}^{\vee}(q)(\vec{v}) = 0 = \mathcal{H}^{\vee}(f)(\vec{v})$. When $\vec{v}$ is a representative, we have

$$\mathcal{H}^{\vee}(f) = \bigvee_{\mathcal{H}(\vec{v'})=v} f(\vec{v'}) = \bigvee_{\mathcal{H}(\vec{v'})=v} (p(\vec{v'}) \wedge q(\vec{v'}))$$

Likewise,

$$\mathcal{H}^{\vee}(p)(\vec{v}) \wedge \mathcal{H}^{\vee}(q)(\vec{v}) = \bigvee_{\mathcal{H}(\vec{v'})=v} p(\vec{v'}) \wedge \bigvee_{\mathcal{H}(\vec{v'})=v} q(\vec{v'})$$

It is easy to prove that $\bigvee_i (a_i \wedge b_i) \rightarrow (\bigvee_i a_i) \wedge (\bigvee_i b_i)$ is a tautology. Therefore,

$$\bigvee_{\mathcal{H}(\vec{v'})=v} (p(\vec{v'}) \wedge q(\vec{v'})) \rightarrow \bigvee_{\mathcal{H}(\vec{v'})=v} p(\vec{v'}) \wedge \bigvee_{\mathcal{H}(\vec{v'})=v} q(\vec{v'}).$$

Consequently, $\mathcal{H}^{\vee}(f)(\vec{v}) \rightarrow \mathcal{H}^{\vee}(p)(\vec{v}) \wedge \mathcal{H}^{\vee}(q)(\vec{v})$. In general, $\mathcal{H}^{\vee}(f) \rightarrow \mathcal{H}^{\vee}(p) \wedge \mathcal{H}^{\vee}(q)$. $\square$

The $\vee$-type aBDDs can be used to compute existential abstraction which is defined in Chapter 2.4 according to the following lemma.

**Lemma 5.4.2** *Given a Boolean function $f(x)$ and an abstraction function $h : D \to A$ and its corresponding auto-abstraction function $\mathcal{H} : D \to D$, the following formula holds*

$$\mathcal{H}^{\vee}(f)(y) = \exists x[\mathcal{H}(x) = y \wedge f(x)]$$

**Proof** We prove this lemma by looking at two conditions.

1. $y$ is a representative, i.e., exists $x \in D, \mathcal{H}(x) = y$. Then

$$\mathcal{H}^{\vee}(f)(y) = \bigvee_{\mathcal{H}(x)=y} f(x) = \exists x[\mathcal{H}(x) = y \wedge f(x)].$$

2. $y$ is not a representative, i.e., there is no $x \in D, \mathcal{H}(x) = y$. In other words, $\mathcal{H}(x) = y$ is always false. Therefore, $\exists x[\mathcal{H}(x) = y \wedge f(x)]$ is always false. According to the definition of $\vee$-type aBDDs, $\mathcal{H}^{\vee}(f)(y) = 0$ in this case.

Overall, the lemma holds. □

Similarly to S-type, 0-type or 1-type aBDDs, $\vee$-aBDDs can be easily extended to deal with multiple abstraction functions. For a Boolean function $f : D^n \to B$, the $\vee$-type aBDD $\mathcal{H}(f)$ is defined as

$$\mathcal{H}^{\vee}(f)(x_1, \cdots, x_n) = \begin{cases} \bigvee f(y_1, \cdots, y_n) & x_1 = \mathcal{H}(y_1), \cdots, x_n = \mathcal{H}(y_n) \\ 0 & \text{otherwise} \end{cases}$$

Vectors $x_i, y_i \in D$, and $\mathcal{H}(y_i) = x_i$ implies that $x_i$ is the representative in the equivalence class of $y_i$. It is easy to prove that Lemma 5.4.1 holds for multiple abstraction functions. Moreover, we have

**Lemma 5.4.3** *Given a Boolean function $f(x_1, \cdots, x_n)$ and an abstraction function $\mathcal{H} : D \to D$, the following formula holds*

$$\mathcal{H}^{\vee}(f) = \exists x_1, \cdots, x_n[\mathcal{H}(x_1) = y_1 \wedge \cdots \wedge \mathcal{H}(x_n) = y_n \wedge f(x)]$$

This lemma will be the foundation for the results of Chapter 6.3.

## 5.5 Summary

In the previous three sections, we discussed four types of aBDDs for Boolean functions. We also show that each of these four types of aBDDs can be extended to the case of multiple abstraction functions as well. As a summary, the definition of these four types of aBDDs are described in the following table.

| | |
|---|---|
| S-type: | $\mathcal{H}^s(f)(x_1,\ldots,x_n) = f(\mathcal{H}(x_1),\ldots,\mathcal{H}(x_n))$ |
| 0-type: | $\mathcal{H}^0(f)(x_1,\ldots,x_n) = \begin{cases} f(x_1,\ldots,x_n) & \mathcal{H}(x_1)=x_1,\ldots \\ 0 & \text{otherwise} \end{cases}$ |
| 1-type: | $\mathcal{H}^1(f)(x_1,\ldots,x_n) = \begin{cases} f(x_1,\ldots,x_n) & \mathcal{H}(x_1)=x_1,\ldots \\ 1 & \text{otherwise} \end{cases}$ |
| $\vee$-type: | $\mathcal{H}^\vee(f)(x_1,\cdots,x_n) = \begin{cases} \bigvee f(y_1,\cdots,y_n) & x_1=\mathcal{H}(y_1),\ldots \\ 0 & \text{otherwise} \end{cases}$ |

Here, $x_i$ and $y_i$ denote vectors in $D$, and $\mathcal{H}(y_i) = x_i$ means that $x_i$ is the representative of the equivalence class of $y_i$. The properties and algorithms given for a single abstraction can be easily generalized for the multiple abstractions case.

Given an abstraction function $h$, the auto-abstraction function $\mathcal{H}$ determines the representatives and non-representatives. Selecting different functions for representatives and non-representatives results in different types aBDDs. In Table 5.5, we summarize the properties of each type of aBDDs.

| aBDDs Types | Rep Nodes | NonRep Nodes | Distribute Over Ops | Algorithm |
|---|---|---|---|---|
| S-type | kept | changed | yes for any | DFS based |
| 0-type | kept | changed | yes for $\wedge$,$\vee$ | DFS based |
| 1-type | kept | changed | yes for $\wedge$,$\vee$ | DFS based |
| $\vee$-type | changed | changed | no except $\vee$ | BFS based |

In the next chapter, we will discuss how to apply different types of aBDDs in different situations.

# Chapter 6

# Applications of abstract BDDs

In this chapter, we will discuss three applications of abstract BDDs: equivalence checking, dynamic reordering and model checking.

## 6.1 Equivalence checking using abstract BDDs

For many abstraction functions, aBDDs are usually much smaller than the BDDs of the original function. Moreover, aBDDs maintain partial information of the original Boolean functions. They can be used as a sufficient condition to check *in*-equivalence of large combinational circuits.

**Lemma 6.1.1** *Given two Boolean functions* $f, g : D^n \to B$ *where* $D = B^k$*,*

$$\begin{array}{c}\mathcal{H}^s(f) \neq \mathcal{H}^s(g) \to f \neq g \\ \mathcal{H}^0(f) \neq \mathcal{H}^0(g) \to f \neq g \\ \mathcal{H}^1(f) \neq \mathcal{H}^1(g) \to f \neq g \\ \mathcal{H}^\vee(f) \neq \mathcal{H}^\vee(g) \to f \neq g\end{array}$$

**Proof** Let us assume that $\mathcal{H}^s(f) \neq \mathcal{H}^s(g)$. Therefore, there must exist $\vec{v_1}, \ldots, \vec{v_n} \in D$, where

$$\mathcal{H}^s(f)(\vec{v_1}, \ldots, \vec{v_n}) \neq \mathcal{H}^s(g)(\vec{v_1}, \ldots, \vec{v_n}).$$

According to the definition,

$$\begin{array}{c}\mathcal{H}^s(f)(\vec{v_1}, \ldots, \vec{v_n}) = f(\mathcal{H}^s(\vec{v_1}), \ldots, \mathcal{H}^s(\vec{v_n})) \\ \mathcal{H}^s(g)(\vec{v_1}, \ldots, \vec{v_n}) = g(\mathcal{H}^s(\vec{v_1}), \ldots, \mathcal{H}^s(\vec{v_n})),\end{array}$$

therefore,

$$f(\mathcal{H}^s(\vec{v_1}), \ldots, \mathcal{H}^s(\vec{v_n})) \neq g(\mathcal{H}^s(\vec{v_1}), \ldots, \mathcal{H}^s(\vec{v_n}))$$

Generally, $f \neq g$. Using the similar argument, we can easily prove that the rest three formulas also hold. $\Box$

When the BDDs for a large multiple-output combinational circuit become extremely large, aBDDs can usually be built instead.

The equivalence checking procedure using aBDDs is sketched as follows.

1. Given a circuit, choose a set of appropriate abstraction functions.

2. Select an abstraction function $h$ out of a set of abstraction functions. This set will be provided based on the nature of the circuit.

3. Build S-type (0-type or 1-type) aBDDs for the specification and the implementation circuit using the abstraction function $h$.

4. For S-type aBDDs, directly compare the two aBDDs obtained for specification and implementation. For 0-type or 1-type aBDDs, apply abstraction once again and then compare the two obtained aBDDs. If they are different, an error is detected. Otherwise, choose a different abstraction function from the set and repeat step 3 with a different abstraction function.

In general, there is no procedure to select a set of abstraction functions that will detect all the errors in a circuit. Nevertheless, we believe that our methodology can be extremely useful in practice, since an initial design is much more likely to contain errors than to be correct.

We use a simple example to illustrate our algorithm. Assume that we have an abstraction function $h$ for the circuit in Figure 6.1. Assume that we have built the BDDs for $p$ and $q$. By performing S-type abstraction on them, we obtained $\mathcal{H}^s(p)$ and $\mathcal{H}^s(q)$. According to Lemma 5.2.1, $\mathcal{H}^s(g) = \neg[\mathcal{H}^s(p) \wedge \mathcal{H}^s(q)]$. Therefore, we can obtain an S-type aBDD for $g$ using ordinary logic operations. On the other hand, the procedure is slightly different for 0(1)-type aBDDs. According to Lemma 5.3.1, $\mathcal{H}^0(g) = \mathcal{H}^0(\neg[\mathcal{H}^0(p) \wedge \mathcal{H}^0(q)])$ and $\mathcal{H}^1(g) = \mathcal{H}^1(\neg[\mathcal{H}^1(p) \wedge \mathcal{H}^1(q)])$. In order to obtain the 0(1)-type aBDDs, abstraction has to be applied again after the logic operations.

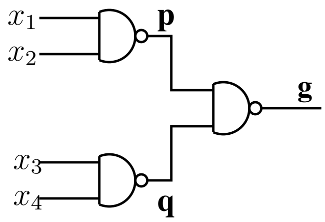



Figure 6.1: A simple combinational circuit

We have implemented S-type and 0-type aBDDs into the CMU BDD package and performed two sets of experiments on the ISCAS85 benchmark circuits using both S-type and 0-type aBDDs. Design errors in the circuit were injected one by one by selecting a stuck-at fault on one input of an arbitrary gate. In the first experiments (Table 6.1,6.2), we use S-type aBDDs and disable the dynamic reordering. We choose $D = B^{\#var}$, i.e., all the Boolean variables are mapped as one symbolic variables. The experiments using two different abstraction functions: $h_1(x) = count1(x)$ and $h_2(x) = x \bmod p$ where $p$ is a prime number. In Table 6.1,6.2 , *Det Errs* is the number of faults detected by these three methods, and *Max # Nodes* is the maximum number of BDD nodes that need to be held in memory, which is usually much larger than the final BDD size. *Avg.Time* is the average time to detect a design error. The

OBDD results for c2670, c5315, c6288 and c7552 are not reported because they exceeded the memory limit.

| circuits | Errs | Det Errs | | |
|---|---|---|---|---|
| | | OBDD | Symm | Resid |
| c432 | 50 | 50 | 50 | 33 |
| c499 | 50 | 50 | 40 | 28 |
| c880 | 50 | 50 | 28 | 7 |
| c1355 | 50 | 50 | 40 | 28 |
| c1908 | 50 | 48 | 40 | 36 |
| c2670 | 10 | unable | 5 | 2 |
| c3540 | 50 | 50 | 24 | 16 |
| c5315 | 10 | unable | 10 | 3 |
| c6288 | 10 | unable | 6 | 6 |
| c7552 | 10 | unable | 9 | 10 |

Table 6.1: Number of detected errors using OBDDs and aBDDs

| circuits | Max # Nodes | | | Avg.Time | | |
|---|---|---|---|---|---|---|
| | OBDD | Symm | Resid | OBDD | Symm | Resid |
| c432 | 4712 | 4604 | 3902 | 1.15 | 7.94 | 19.70 |
| c499 | 95745 | 9481 | 27121 | 22.74 | 16.72 | 48.64 |
| c880 | 637338 | 7705 | 4999 | 138.25 | 58.17 | 180.56 |
| c1355 | 96357 | 9497 | 27476 | 25.49 | 44.93 | 129.48 |
| c1908 | 70196 | 6274 | 15838 | 35.95 | 22.82 | 61.86 |
| c2670 | – | 132593 | 774009 | – | 5449.37 | 5073.17 |
| c3540 | 1522988 | 9927 | 8267 | 299.89 | 109.61 | 379.06 |
| c5315 | – | 208795 | 234716 | – | 4618.01 | 10052.3 |
| c6288 | – | 7317 | 38 | – | 86.52 | 61.20 |
| c7552 | – | 366462 | 2301523 | – | 11963.65 | 18405.8 |

Table 6.2: BDD overhead and Average time used

In the second experiments, we use 0-type aBDDs with dynamic reording. We choose $D = B^{10}$ and the abstraction function $h(x_1) = count1(x_1)$. The abstraction function for the other variables is the identity abstraction function, i.e., $h(x_i) = x_i$ for $i > 1$.

| Circuit | Errs | Detected Errors | | BDD Size | |
|---|---|---|---|---|---|
| | | OBDD | aBDD | OBDD | 0-type |
| c432 | 10 | 10 | 10 | 4403 | 3307 |
| c499 | 10 | 10 | 10 | 16850 | 12767 |
| c880 | 10 | 10 | 10 | 12905 | 5077 |
| c1355 | 10 | 10 | 10 | 28582 | 16661 |
| c1908 | 10 | 10 | 9 | 13153 | 7737 |
| c2670 | 10 | 10 | 10 | 28926 | 16643 |
| c3540 | 10 | 9 | 5 | 67586 | 12790 |
| c5315 | 10 | 10 | 10 | 17905 | 13677 |
| c6288 | 10 | – | 10 | — | 8128 |
| c7552 | 10 | 10 | 10 | 13637 | 16889 |

In this experiment, the aBDD based approach can detect over 90$\%$ of the errors.

## 6.2 Improving variable ordering using 0-type abstract BDDs

Let $f : B^m \to B$ be a Boolean function over the Boolean variables $y_1, y_2, \ldots, y_m$. A *cube* $c_i$ is a monomial over the variables $y_1, \cdots, y_k$ $(k < m)$. *Cube based sampling* [60] partitions the domain of $f$ into smaller cubes $c_1, \cdots, c_{2^{m-k}}$ and uses dynamic variable ordering to select a good ordering for the restriction $f_i = f \wedge c_i$. The ordering for $f$ is obtained by combining the orderings of several randomly chosen $f_i$. The quality of the resulting ordering may not be very good if $f_i$ does not closely approximate $f$. Thus, if the subset of cubes is selected randomly, there may be significant variance in the approximations. Consequently, the final ordering for $f$ may not be good.

This problem can be alleviated by using a new sampling technique. Instead of analyzing *one* random cube, we automatically consider multiple cubes at the same time by using 0-type aBDDs. We call this new technique *window based sampling*.

Intuitively, a window is a union of some number of cubes. Assume that we choose $t$ disjoint windows $w_1, \ldots, w_t$. Hence, we can partition $f$ into

$f_1, \ldots, f_t$, where $f_i = f \wedge w_i$. In our window based approach we choose the sampling windows using 0-type aBDDs.

According to our definition, we know that $x_1 \in D = B^k$. Setting $n = \lfloor m/k \rfloor + 1$, we can rewrite the Boolean function $f(y_1, \ldots, y_m)$ as $f(x_1, \ldots, x_n)$, where $x_i = \langle y_{(i-1)k+1}, \ldots, y_{i*k} \rangle$ for $i = 1, \ldots, n-1$ and $x_n = \langle y_{(n-1)k+1}, \ldots, y_m, 0 \ldots 0 \rangle$. Let $d = \langle a_1, \ldots, a_k \rangle$ be a $k$-bit vector where $a_i \in B$ and $d \in D = B^k$. It is easy to see that $d$ induces a cube $c_d$ where

$$c_d = \bigwedge_{i=1}^{k} \begin{cases} y_i & a_i = 1 \\ \neg y_i & a_i = 0 \end{cases}$$

Given an abstraction function $h$, assume that $h_1 = h$ and $h_i(x_i) = x_i$ for $i > 1$. In other words, abstraction is only applied to the top variable $x_1$. The other variables are kept unabstracted. If we define a window $w_{\mathcal{H}} = \cup_{\vec{u}=\mathcal{H}(\vec{u})} c_{\vec{u}}$, then the following lemma holds.

**Lemma 6.2.1** *Let $f$ be a Boolean function and $\mathcal{H}$ be an auto-abstraction function. Then $\mathcal{H}^0(f) = f \wedge w_{\mathcal{H}}$.*

**Proof** Note that $h_i$ ($i > 1$) is identity function. Therefore, the corresponding auto-abstraction function $\mathcal{H}_i$ is also identity function. Therefore, the 0-type aBDD for $f$ can be defined as

$$\mathcal{H}^0(f)(x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n) & \mathcal{H}(x_1) = x_1 \\ 0 & \text{otherwise} \end{cases}$$

Or, it can also be written as

$$\mathcal{H}^0(f)(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \wedge (\bigvee_{d=\mathcal{H}(d)} x_1 = d).$$

It is easy to see that $x_1 = d$ is the same as $c_d$. Therefore, $\mathcal{H}^0(f) = f \wedge w_{\mathcal{H}}$. □

According to Lemma 6.2.1, using 0-type aBDDs provides a natural way to implement the window based sampling method. First, we select a set of *control* *variables*. These variables are heuristically determined by traversing the circuit in a depth-first order where nodes are selected so that the distance from a node to the primary inputs is minimized. Next, we choose an abstraction function for a set of control variables and build an abstract BDD for the function $f$ with dynamic reordering. Since this abstract BDD partially captures the functionality of $f$, a good ordering for the abstract BDD is likely to be a good ordering for $f$ as well. Different abstraction functions usually produce different orders. From our experiments, we have found that the *count1* function $\sum_{i=1}^{k} y_i$ and the *logarithmic* abstraction function $\lfloor \log_2 \sum_{i=1}^{k}(2^i y_i) \rfloor$ are good choices. Note that these abstraction functions are parameterized by the number of variables. In each case, the number of cubes is relatively small. For example, a count1 abstraction function on $k$ variables determines $k+1$ cubes.

Our method has 4 steps: the estimation phase, the candidate-order selection phase, the testing phase (circuit filter phase), and the evolution phase. These 4 phases produce an initial ordering for building the final BDD and are described below.

**Step 1.** In the *estimation phase*, we try $j$ different abstraction functions and determine the number of variables in each. Starting from the top variable, we choose the set of abstracted variables $k_i (i \leq j)$ incrementally. For each cube in the window given by the abstraction function, we partially simulate the circuit. We choose $k_i$ to be the size of the abstraction function if simulating one of the cubes greatly decreases the number of gates left in the circuit.

**Step 2.** In the *candidate-order selection phase*, we apply $j$ different abstraction functions to the top $k_i (i < j)$ variables selected in the previous phase. Then, we build the aBDDs for the original Boolean function with dynamic reordering. Each produces a new variable ordering. In our experiments, we choose $j$ to be

2 or 3 and use the subsequent phases to reject and refine these orderings.

**Step 3.** The purpose of the *circuit filter phase* is to filter out the bad orderings. We estimate the quality of a given variable ordering by building the BDD with this ordering up to a certain target gate inside the circuit (with dynamic reordering disabled). An obvious question is how we choose the target gate. Using some threshold level, we pick the gate between the primary inputs and this threshold level whose cone covers the maximum number of primary inputs. The intuition for this step is that we want to consider as many variables as possible to compare the orderings for all of the variables obtained from Step 2.

**Step 4.** After filtering out the bad orderings, we use the *evolution filter* to decide which is the best ordering from the ones that remain. Using another window defined by a new abstraction function, we build aBDDs for the remaining orderings obtained from Step 3. We choose the ordering which has the minimum number of BDD nodes as our final order. This idea is also discussed in [60]. The difference is that we select windows using abstract BDDs instead of randomly selected cubes.

In a cube based sampling technique, since only one cube is considered at a given time, a sample may map to a trivial function. A window based sampling method considers a large number of cubes at one time; it is highly unlikely that each of these cubes will reduce to a trivial function. Thus, even if random cubes were generated, a window based sampling is far more stable.

A function sampled using windows effectively contains a restriction of the original function on each of the cubes. Thus, when we reorder our sampled function, we are implicitly trying to produce an order which is simultaneously "good" for each of these restrictions. Intuitively, this is important because a variable order produced from restriction by any single cube may not be good for the whole function. Considering multiple cubes at the same time and "averaging" their effect is more likely to produce better results. For many circuits we find that the variable order produced by using windows is far better than the order produced by cubes.

The advantages of a window based method are particularly impressive when a single order is needed for all outputs of a multiple-output circuit. In fact, if we rely on cubes alone, then since the control variables may differ for different outputs, the cubes effective for one output may not yield good results for another. Techniques based on aBDDs are far superior in this case as well.

Our experiments are performed on a 360MHz Sun UltraSparc-60 with 512Mb RAM using the CUDD-2.2.0 package. In our tables, BDD size is measured by the number of BDD nodes. Runtime entries refer to the time taken for the sampling phases, as well as the time taken to construct the final BDD from the order computed by sampling. The "DFS-MIN" entries refer to the DFS based static variable ordering method described earlier. Similarly, all CUDD entries refer to CUDD-2.2.0 using *sift*, except for "CUDD SiftConv" which was obtained by replacing *sift* with *sift-convergence* throughout the experiment. The "Using aBDD" column refers to the sampling technique which uses 0-type abstract BDDs. We conducted two sets of experiments. The first experiment shows how the technique behaves on single output functions, while the second experiment deals with multiple output functions.

Note that our aBDD method gives deterministic results (unlike [60]). For this purpose, in our experiments, we use two abstraction functions: the *count1* function and the *logarithmic* functions (see Chapter 5.2).

**Experiment 1** (Table 6.3, and Figure 6.2): First, we use the order computed by sampling to build the BDD statically. Except for slightly inferior

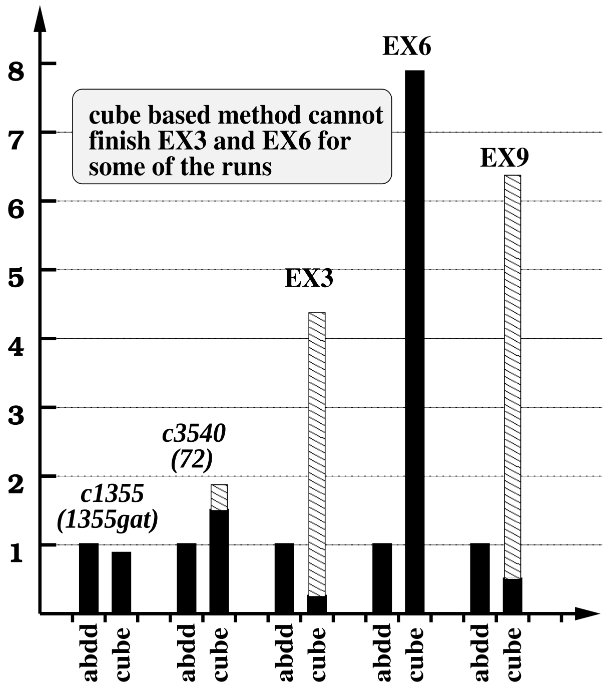



Figure 6.2: Static ordering with aBDD vs. cube based method

orderings on c499 and c1355 (both circuits are functionally equivalent) we find that our methods always produce better variable orderings than those produced by DFS search based static techniques (Table 6.3). For many industrial examples we find that DFS-MIN cannot even process the circuits. Interestingly, for c3540 and EX1, we find that our static order using abstract BDD based windows is better than even the dynamic ordering obtained using the CUDD-2.2.0 package, and for EX6, comparable. Thus, we believe that our window based sampling method is superior to other static ordering methods in terms of efficiency as well as stability.

Figure 2 gives some representative data for comparing the performance of static ordering methods that use an initial ordering provided by cube based sampling vs. window based sampling using aBDDs. It is easy to see that cube based method suffers from very large variance. However, since window based sampling is deterministic, there is no variance at all. Interestingly, for EX3 and

| | **BDD Nodes** | | **TIME (sec)** | |
|---|---|---|---|---|
| Ckts | **DFS -MIN** | **Using aBDDs** | **DFS -MIN** | **Using aBDDs** |
| c432 | 5624 | 3956 | 1.6 | 3.1 |
| c499 | 3466 | 3429 | 0.1 | 5.1 |
| c1355 | 3652 | 3109 | 0.1 | 5.0 |
| c1908 | 2187 | 1428 | 0.2 | 2.6 |
| **c3540** | **55730** | **6976** | **9.1** | **30** |
| c6288 | 19417 | 22360 | 5.1 | 132 |
| c6288 | 48483 | 42781 | 17.0 | 127 |
| **EX1** | **fail** | **2779448** | **fail** | **111** |
| **EX2** | **881339** | **596415** | **9.5** | **24** |
| EX3 | 966210 | 738906 | 8.8 | 91 |
| **EX6** | **fail** | **20994** | **fail** | **134** |
| **EX8** | **fail** | **34918** | **fail** | **89** |
| **EX10** | **fail** | **942** | **fail** | **88** |

Table 6.3: Static order using sampling with aBDDs

EX6, aBDD based methods can create a small BDD for the output function, but cube based sampling fails for some of the runs!

**Experiment 2** (Table 6.4 and Figure 6.3) demonstrates the utility of window based sampling in a dynamic variable ordering scheme. That is, we show how sifting based reordering techniques can be significantly improved if they are supplied with an initial variable ordering generated using a window based sampling technique. In Table 6.4, we find that we can produce far smaller graphs than the traditional dynamic reordering methods (*sift, sift-convergence*). Also, for most of the large circuits we take less time. Sometimes, the difference is dramatic; in EX3 we take almost an order of magnitude less space and 6 times less runtime. Compared with sampling approaches, our method is also superior (Figure 6.3) since our method does not have the large deviation problem.

**Experiment 3** (Table 6.2): We performed another set of experiments to verify the efficiency of window based methods on multiple output functions. It is

| Ckts | SPACE (# of BDD Nodes) CUDD Sift | CUDD SiftConv | Using aBDDs | TIME (in seconds) CUDD Sift | CUDD SiftConv | Using aBDDs |
|---|---|---|---|---|---|---|
| c432 | 379 | 377 | 367 | 1.3 | 2.8 | 2.9 |
| c499 | 3457 | 3650 | 3117 | 3.5 | 7.2 | 5.3 |
| c1355 | 2557 | 3337 | 3529 | 3.2 | 11.0 | 6.9 |
| c1908 | **901** | **758** | **763** | **2.0** | **4.5** | **2.6** |
| c3540 | **8045** | **5486** | **5510** | **46.0** | **54.0** | **31.0** |
| c6288 | 16774 | 16693 | 16746 | 40.0 | 110.0 | 56.0 |
| c6288 | 40024 | 39942 | 40024 | 88.0 | 251.0 | 103.0 |
| EX1 | **1467** | **644** | **748** | **41** | **89** | **33** |
| EX2 | **13390** | **14771** | **9431** | **22** | **98** | **33** |
| EX3 | **633780** | **655556** | **63404** | **1320** | **6780** | **230** |
| EX4 | **163854** | **fail** | **130589** | **3535** | **fail** | **2667** |
| EX5 | **190674** | **190674** | **63916** | **2616** | **2586** | **480** |
| EX6 | **20343** | **15905** | **13457** | **146** | **334** | **120** |
| EX7 | **118378** | **67384** | **40698** | **522** | **517** | **191** |
| EX8 | **289619** | **387116** | **186754** | **786** | **4781** | **1365** |

Table 6.4: Deterministic sampling using aBDD

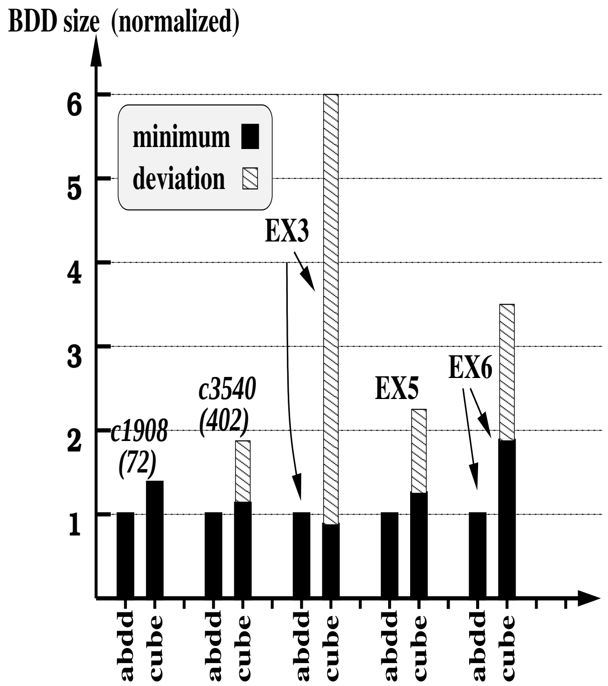



Figure 6.3: Dynamic ordering with aBDD vs. cube based method

known that sifting works very well for ISCAS85 circuits [85] and for many circuits, there may not be scope for significant improvement. However, our approach still outperforms CUDD for some of the circuits (c1908 and c7552). For large industrial circuits, our approach is definitely much better than CUDD (*sift*) with respect to both time and space.

| | **CUDD Sift** | | **Using aBDDs** | |
|---|---|---|---|---|
| Ckts | **BDD** Size | **CPU Time** | **BDD** Size | **CPU Time** |
| c432 | 1246 | 0:02 | 1224 | 0:03 |
| c499 | 25897 | 0:29 | 26798 | 1:03 |
| c880 | 4821 | 0:06 | 4463 | 0:06 |
| c1355 | 25897 | 0:31 | 26579 | 0:56 |
| c1908 | **9102** | **0:07** | **5946** | **0:08** |
| c2670 | 2412 | 0:15 | 3070 | 0:31 |
| c3540 | 23857 | 0:27 | 24122 | 1:02 |
| c5315 | 2108 | 0:06 | 2712 | 0:07 |
| c7552 | **18363** | **2:26** | **7206** | **0:59** |
| M1 | **2595K** | **1:54:45** | **1866K** | **1:26:41** |
| M2 | **4283K** | **8:36:00** | **4120K** | **2:50:15** |
| M3 | **963K** | **1:17:15** | **487K** | **28:49** |
| M4 | **fail** | **fail** | **2195K** | **1:13:26** |
| M5 | **5976** | **0:48** | **1568** | **2:23** |
| M6 | **89639** | **4:24** | **13625** | **2:36** |

Table 6.5: Sampling using 0-type aBDD for ISCAS85 circuits

## 6.3 Model Checking Using $\vee$-type Abstract BDDs

In this section, we will discuss how to use $\vee$-type aBDDs to construct an abstract Kripke structure and applications.

### 6.3.1 Abstraction for $\mathrm{ACTL}^{\star}$

Given a structure $M = (S, S_0, R)$, where $S$ is the set of states, $S_0 \subseteq S$ is the set of initial states, and $R \subseteq S \times S$ is the transition relation, the abstract structure

$M_h = (S_h, S_{0,h}, R_h)$ is defined as follows:

$$\begin{array}{lcl} S_{0,h} & = & \exists x_1 \cdots x_n [h(x_1) = \hat{x}_1 \wedge \cdots \wedge h(x_n) = \hat{x}_n \wedge S_0] \\ R_h & = & \exists x_1 \cdots \exists x'_1 \cdots [h(x_1) = \hat{x}_1 \wedge \cdots \wedge h(x'_1) = \hat{x'}_1 \wedge \cdots \wedge R] \end{array}$$

where $S = D^n$ and $x_i$ is a symbolic variable over $D$. On the other hand, we define another abstract structure $M_{\mathcal{H}} = (S_{\mathcal{H}}, S_{0,\mathcal{H}}, R_{\mathcal{H}})$, which we construct using $\vee$-type aBDDs:

- The *state set* $S_{\mathcal{H}}$ is the image of $S$ under the auto-abstraction function $\mathcal{H}$.
- The *initial set of states* $S_{0,\mathcal{H}}$ is the image of $S_0$ under the function $\mathcal{H}$. Notice that if $S_0$ is represented as a boolean function, then $S_{0,\mathcal{H}}$ corresponds to the $\vee$-type aBDD $\mathcal{H}(S_0)$ (see Lemma 5.4.3).
- The *transition relation* $R_{\mathcal{H}}$ is the image of $R$ under the function $\mathcal{H}$. Notice that if $R$ is represented as a boolean function, then $R_{\mathcal{H}}$ corresponds to the $\vee$-type aBDD $\mathcal{H}(R)$ (see Lemma 5.4.3).

In Chapter 2.4, Theoerem 2.4.2 states that $M_h$ and $M_{\mathcal{H}}$ are *isomorphic structures*, i.e., there is a bijection $u : S_h \to S_{\mathcal{H}}$ such that $u(S_0) = S_{0,\mathcal{H}}$ and $u(R_h) = R_{\mathcal{H}}$.

According to the definition of existential abstraction, the standard *relational product* technique can be used to build the abstract transition relation $R_h$. We call this straightforward approach the traditional approach or method. Using $\vee$-type aBDDs has advantages over the traditional approach. First, in the traditional method the BDD for the abstraction functions has to be constructed before applying the method. For many abstraction functions, these BDDs are very hard to build. Second, in our experience a good variable ordering for an abstraction function might be different from a good variable ordering for the transition relation of the system, but standard model checkers would enforce them to coincide. Our approach using abstract BDDs does not suffer from these problems since we never explicitly build the BDDs for the abstraction functions [27]. Abstraction functions are employed while building the abstract BDD corresponding to the transition relation.

In order to test our ideas we modified the model-checker SMV. In our implementation, the user gives an abstraction function for each variable of interest. Once the user provides a system model and the abstraction functions, our method is completely automatic. We consider two examples in this paper: a pipelined multiplier design and the PCI local bus protocol.

### 6.3.2 Case studies

**Verification of a pipelined multiplier** In [31], Clarke, Grumberg, and Long propose an approach based on the *Chinese Remainder Theorem* for verifying *sequential* multipliers. The statement of the *Chinese Remainder Theorem* can be found in most texts on elementary number theory and will not be repeated here. Clarke, Grumberg, and Long use the modulus function $h(i) = i \bmod m$ for abstraction. They exploit the distributive property of the modulus function over addition, subtraction, and multiplication.

$$((i \bmod m) + (j \bmod m)) \bmod m \equiv (i + j) \bmod m$$

$$((i \bmod m) - (j \bmod m)) \bmod m \equiv (i - j) \bmod m$$

$$((i \bmod m) \times (j \bmod m)) \bmod m \equiv (i \times j) \bmod m$$

Let $\bullet$ represent the operation corresponding to the *implementation*. The goal is to prove that $\bullet$ is actually multiplication $\times$, or, in other words, for all $x$ and $y$ (within some finite range) $x \bullet y$ is equal to $x \times y$. If the actual implementation of the multiplier is composed of *shift-add* components, then the modulus function will distribute over the $\bullet$ operation. Therefore, we have the following equation:

$$(x \bullet y) \bmod m \;=\; [(x \bmod m) \bullet (y \bmod m)] \bmod m$$

Using this property and the Chinese Remainder Theorem, Clarke, Grumberg, and Long verify a sequential multiplier.

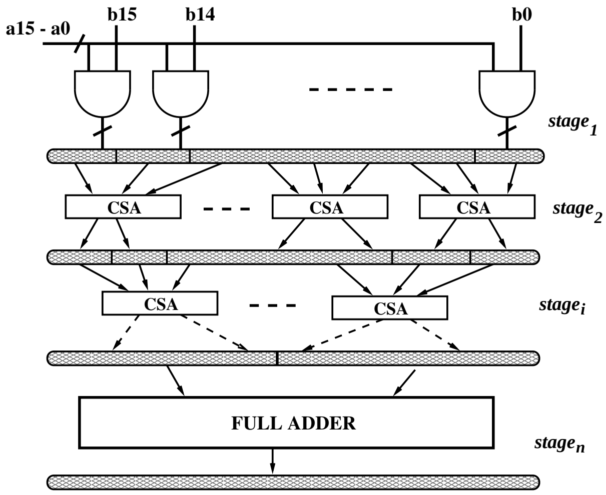



Figure 6.4: Carry-save-adder pipeline multiplier

Unfortunately, this approach may not work if the multiplier is not composed of shift-add components. Suppose there is a mistake in the design of the multiplier, then there is no guarantee that the modulus operator will distribute over the operation $\bullet$ (corresponding to the actual implementation). For example, the mistake might scramble the inputs in some arbitrary way which breaks the distributive property of the $\bullet$ operation. In this case, the method proposed by Clarke, Grumberg and Long is not complete and may miss some errors. Therefore, before we apply the methodology in [31] it is necessary to check the distributive property of the modulus function with respect to the $\bullet$ operator. In other words, we must show that the following equation holds:

$$(x \bullet y) \bmod m \;=\; [(x \bmod m) \bullet (y \bmod m)] \bmod m$$

We illustrate our ideas by verifying a $16 \times 16$ pipelined multiplier which uses carry-save adders (see Figure 6.4). Notice that the first stage consists of *shift* operations and the last stage corresponds to the *add* operation. It easy to show that the first and the last stages satisfy the distributive property. In fact, this can be determined using classical equivalence checking methods. We will focus our attention on the intermediate stages.

Notice that the Chinese Remainder Theorem implies that it is enough to verify the multiplier by choosing $m = 5, 7, 9, 11, 13, 16, 17, 19, 23$ because of the following equation:

$$5 * 7 * 9 * 11 * 13 * 16 * 17 * 19 * 23 = 5354228880 > 2^{32} = 4294967296.$$

Our technique works as follows:

- First verify that each pipelined stage satisfies the distributive property using numbers in the set $\{5, 7, 9, 11, 13, 16, 17, 19, 23\}$). Formally, let $\bullet_i$ correspond to the operation of the $i$-th stage in the pipeline. We want to verify the following equation for all $m$ in the set $\{5, 7, 9, 11, 13, 16, 17, 19, 23\}$ and $1 \leq i \leq 6$:

  $$(x \bullet_i y) \bmod m \quad = \quad (x \bmod m \bullet_i y \bmod m) \bmod m$$

  If the equation given above is violated, we have found a error. Notice that the equation given above can be checked by building the abstract BDD for the transition relation corresponding to the $i$-th stage.

- Next, assume that all the pipelined stages satisfy the distributive property. In this case, we can apply the method proposed by Clarke, Grumberg, and Long because the entire design will also satisfy the distributive property.

In Figure 6.5 we give our experimental results for the first step. The row for space usage corresponds to the largest amount of memory that is used during verification.

| modulus | 5 | 7 | 9 | 11 | 13 | 16 | 17 | 19 | 23 |
|---|---|---|---|---|---|---|---|---|---|
| time(s) | 99 | 137 | 199 | 372 | 636 | 130 | 1497 | 2648 | 6977 |
| space(MB) | 7.7 | 12.8 | 21.5 | 51.7 | 92.5 | 9.2 | 210 | 231 | 430 |

Figure 6.5: Experimental Results for various modulus

We detected a number of actual errors in our design using this technique. All of the errors that we found were caught using $m = 3$. The average time and space requirements to find the errors for the different stages are shown in Figure 6.6. The $16 \times 16$ multiplier design could not be verified by SMV without using abstraction. Therefore, we could not measure how much we saved by using our technique.

| stages | st 1 | st 2 | st 3 | st 4 | st 5 | st 6 | total |
|---|---|---|---|---|---|---|---|
| # bugs | 4 | 1 | 1 | 0 | 0 | 1 | 7 |
| avg. time(s) | 5.1 | 4.4 | 3.0 | 2.5 | 1.7 | 1.2 | 89.1 |
| avg. space(M) | 5.6 | 2.7 | 3.7 | 4.8 | 3.6 | 1.7 | 5.96 |

Figure 6.6: Space and Time requirements to find bugs

**Verifying the PCI local bus** The second example we tried is the PCI local bus protocol. During verification, we found a potential error in the PCI bus protocol specification. In particular, we discovered an inconsistency between the textual specification and one of the state machines given in the PCI standard [88]. The precise nature of the error will be explained later.

During model-checking, we used following abstraction functions on various state variables:

- $h(x) = \perp$, where $\perp$ means constant;

- $h(x) =$ if $x \neq 0$ then $1$ else $0$;
- $h(x) =$ if $x > 1$ then $1$ else $0$;

Incidentally, the bug we discovered was *not* found when we applied the techniques proposed in [31].

The PCI Local Bus [88, 89, 100] is a high performance, synchronous bus architecture that can transfer 32-bit or 64-bit data. Its primary goal is to establish an industry standard and optimize for direct silicon (component) interconnection with minimum glue logic required. It supports most processor designs and connects various types of devices on a chip. Bridges are used to extend the PCI bus based systems. There are three types of devices that can be connected to the PCI local bus: masters, target, and bridges. Masters can start transactions. Targets respond to transactions and bridges connect buses. Masters and targets are controlled by a finite-state machine.

A typical PCI bus transaction is demonstrated in Figure 6.7. The request for a transaction starts when a subsystem asserts its request line REQ#. It then waits until being granted the bus by the arbiter by asserting the corresponding GNT# line. This phase is known as the *arbitration phase*. The transaction begins when signal FRAME# is asserted. In the first clock after asserting FRAME#, address is put on the data/address multiplexed lines in the *address phase* and the command lines carry the transaction-type. All target devices listen to this address and if the address maps to their address space, they assert their DEVSEL# lines, indicating they are present on the bus. The master then asserts the signal IRDY#, meaning that it is ready for data transfer. The bus target asserts its TRDY# signal to indicate that the target is ready for data transfer. Data transfer occurs when both IRDY# and TRDY# are asserted, which is known as one *data phase*. A transaction can have more than one *data phase*, and wait cycles can be inserted between data phases by the master (target) by deasserting the IRDY# (TRDY#) signal. One clock cycle before the end of the data transfer phase, the FRAME# signal is deasserted. In the next cycle both IRDY# and TRDY# are deasserted, and the bus goes back to the idle state.

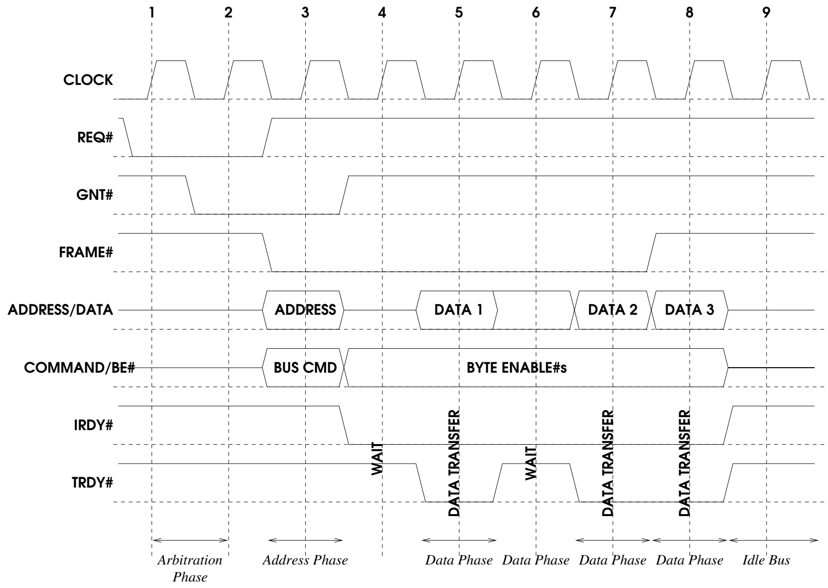



Figure 6.7: A typical PCI bus transaction

In our verification efforts, we considered a simple model which consists of one master, one target, and one bus arbiter. The model includes different timers to meet the timing specification. The master and target both include a *lock* machine to support exclusive read/write. The master also has a data counter to remember the number of data phases.

In the verification, we applied different abstractions to some of the timers, the lock machine and the data counter in the master. We also clustered the transition relations of the major state controllers in both master and the target. We checked various properties dealing with handshaking, read/write transactions, and timing in this simplified model. Next, we describe in detail the property which demonstrates the inconsistency in the design. Description of all the properties that we checked is not given here because of space restrictions.

One of the textual requirements is that *the target responds to every read/write transaction issued by the master*. This important property turns out to be false for the state machine given in the standard when the master reads or writes a single data value. The negation of this property can be expressed in $\mathbf{ACTL}^{\star}$ as follows:

$$\mathbf{AG}(\text{m.req} \wedge \text{m.data\_cnt=1}) \rightarrow \mathbf{A}[(\text{m.req} \wedge \neg\text{t.ack})\mathbf{U}(\text{m.timeout})]) \quad (*)$$

where *m.req* corresponds to the master issuing a transaction; *m.data_cnt=1* means that the master requests one data value; *t.ack* means that the target acknowledges the master's request; and *m.timeout* means that the time the master has allowed for the transaction has expired. If this $\mathbf{ACTL}^{\star}$ formula is true in the abstract model, it is also true in the concrete model. We verified that this formula is true, so there must be an inconsistency in the standard.

The experimental results are shown in Figure 6.8. the first row in Figure 6.8 (*Error*) corresponds to the inconsistency we discovered. The remaining properties are not described here. The second and third columns show the running time and maximum BDD nodes for the original version of SMV. The fourth and fifth columns show the results obtained using our methodology. For some cases our approach reduces the space needed for verification by a factor of 20.

| Properties | SMV | | SMV_ABS | |
|---|---|---|---|---|
| | Time(s) | # nodes | Time(s) | # nodes |
| Error | 278 | 727K | 65 | 33K |
| Property 1 | 20 | 164K | 18 | 14K |
| Property 2 | 137 | 353K | 30 | 66K |
| Property 3 | 99 | 436K | 138 | 54K |
| Property 4 | 185 | 870K | 40 | 36K |
| Property 5 | 67 | 352K | 42 | 57K |

Figure 6.8: Experimental Results for Verifying PCI using Abstraction

### 6.3.3 Abstraction for Variable Ordering

In model checking, the problem of generating a good initial variable ordering is even more serious than the case with combinational circuits. Many static ordering approaches have been proposed [4]. Because the best ordering may change dynamically during the fixpoint computation, these approaches are not powerful enough for many applications. In reality, people generate the initial orders manually or statically and run model checker iteratively to produce a *golden* variable order. This approach is not systematic and may be inefficient for large designs.

Since the abstract Kripke structure describes the basic behavior of the original structure, a good variable order for the abstract structure is likely to be a good ordering for the orginal one. Based on this observation, we propose a new variable ordering scheme as follows:

1. Given a set of abstraction functions, the system automatically builds the abstract Kripke structure using $\vee$-type aBDDs.

2. Next, we run the model checker on the CTL property with the abstract structure with dynamic reordering on. Counterexample generation is disabled in this phase.

3. Finally, we restart the model checker on the original structure using the ordering obtained from the previous step as the initial variable ordering.

As an example, we verified the PCI bus protocol. The PCI local bus protocol includes three types of devices: masters, targets, and bridges. Masters can start transactions, targets respond to transactions, and bridges connect buses. Masters and targets are controlled by finite-state machines. We considered a simple model which consists of one master, one target, and one bus arbiter. The model includes different timers to meet the timing specification. The master and target both include a *lock* machine to support exclusive read/write. The master also has a data counter to support *burst* transactions (multiple data phases). We have observed that the BDD sizes constructed during model checking can be reduced significantly by using the procedure described above. In our verification phase, we have applied abstractions to some of the timers, the lock machine and the data counter in the master. Address and data in both the master and the target are also abstracted. Various properties dealing with handshaking, read/write transactions, and timing are checked in this model. The experimental results are listed in Table 6.6. The initial ordering for both "SMV" and "Using aBDD" columns are provided manually. Obviously, aBDD based approaches are superior to the traditional approach. Note that our approach is totally automatic.

| Property | TIME (sec) | | # Nodes | |
|---|---|---|---|---|
| | **SMV** | **SMV (aBDD Based Order)** | **SMV** | **SMV (aBDD Based Order)** |
| $P_1$ | 542 | 289 | 11984K | 3327K |
| $P_2$ | 242 | 204 | 1778K | 718K |
| $P_3$ | 5882 | 207 | 36077K | 862K |
| $P_4$ | 15 | 77 | 50K | 44K |
| $P_5$ | 424 | 269 | 4458K | 3700K |
| $P_6$ | 179 | 118 | 2472K | 520K |
| $P_7$ | 8970 | 3956 | 28924K | 13964K |
| $P_8$ | 84 | 117 | 645K | 504K |
| $P_9$ | 9946 | 793 | 37288K | 5084K |
| $P_{10}$ | 14 | 75 | 59K | 39K |
| $P_{11}$ | 5580 | 2713 | 20680K | 7850K |
| $P_{12}$ | 293 | 376 | 4632K | 3506K |
| $P_{13}$ | 2043 | 1209 | 19703K | 6002K |
| $P_{14}$ | 2932 | 1862 | 38210K | 17386K |
| $P_{15}$ | 2831 | 118 | 12740K | 520K |
| $P_{16}$ | fail | 3955 | fail | 13964K |
| $P_{17}$ | 63 | 117 | 649K | 504K |

Table 6.6: Generating Initial Ordering for Model Checking using $\vee$-type aB-DDs

# Chapter 7

# Conclusion and Future Work

The state explosion problem is the major problem in applying model checking to real life hardware designs. In this dissertation, I have demonstrated two powerful abstraction techniques to reduce the model size and still prove the property. First, I have showed a counterexample-guided abstraction refinement methodology which is complete for $\mathbf{ACTL}$. Secondly, I have defined a new data structure - abstract BDDs which is a better data structure for building abstract Kripke structure.

One advantage of our counterexample-guided abstraction refinement methodology is that the initial abstraction and the refinement steps are efficient and entirely automatic. All algorithms are symbolic. In comparison to methods like the localization reduction [68], we distinguish more degrees of abstraction for each variable. Thus, the changes in the refinement are potentially finer in our approach. The refinement procedure is guaranteed to eliminate spurious counterexamples while keeping the state space of the abstract model small. These claims are demonstrated by experimental results.

There are many interesting avenues for future research. First, our initial abstraction generation is syntax driven. A semantic driven initial abstraction generation will be more powerful. Using better static analysis techniques on this problem will be very interesting.

Next, it is important to find more efficient approximation algorithms for the NP-complete separation problem encountered during the refinement steps. This will allow us to generate even smaller refined abstract Kripke structures. In the experiments, however, I did not find cases where the described polynomial time computable heuristic works poorly. Identifying such examples may be the starting point of further research.

More fundamentally, the previous research on counterexample-guided refinement including this thesis mostly focusses on searching the *coarsest* refinement. This refinement approach is conservative but not optimal in most cases, because the refinement steps may be smaller than necessary. Looking for optimal refinement will significantly improve the performance of the model checker.

## 7.1 Using other data structures instead of BDDs

The symbolic methods described in this thesis are not tied to representation by BDDs. In particular, no fixpoint computation is involved. Therefore, it is interesting to investigate other approaches including satisfiability based, ATPG based and symbolic simulation approaches. The advantages of using other data structures are as follows.

- In checking spurious counterexample, the original models are typically used. Building transition relations for large models is usually difficult. Sometimes, it is hopeless. However, other data structures do not explicitly build the transition relations.

- Since there is no fixpoint computation during the abstraction and refinement phases, canonicity is not as demanded as for BDD based symbolic model checking. It is possible that satisfiability based techniques can handle significantly larger designs.

## 7.2 Implementing abstraction inside BDDs

In Chapter 5, we introduced a general framework for applying abstraction to BDDs. Abstract BDDs are a first attempt to implement abstraction operations inside BDD packages. By exploiting the nature of abstraction functions, abstract BDDs can improve the performance of abstraction operations significantly.

In this thesis, I described four types of abstract BDDs and their applications. Other types of abstract BDDs can be easily defined following a similar procedure. In my experience, the definition of abstract BDDs should be application-driven.

# Bibliography


[1] Fujitsu aims media processor at DVD. *MicroProcessor Report*, pages 11–13, 1996.

[2] P. A. Abdulla et al. Verification of infinite-state systems by combining abstraction and reachability analysis. In *Computer-Aided Verification*, July 1999.

[3] F. Van Aelten, S. Liao, J. Allen, and S. Devadas. Automatic generation and verification of sufficient correctness properties for synchronous processors. In *International Conference of Computer-Aided Design*, 1992.

[4] A. Aziz, S. Tasiran, and R. K. Brayton. BDD variable ordering for interacting finite state machines. In *Design Automation Conference*, 1994.

[5] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer-Aided Verification*, volume 697 of *LNCS*, pages 29–40, 1993.

[6] Roberto Bayardo, , and Robert Schrag. Using csp look-back techniques to solve exceptionally hard sat instances. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, volume 1118 of *LNCS*, pages 46–60, 1996.

[7] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Computer-Aided Verification*, July 1992.

[8] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Computer-Aided Verification*, June 1998.

[9] S. Berezin et al. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *Formal Methods in Computer-Aided Design*, pages 369–386, 1998.

[10] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference*, pages 317–320, 1999.

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS'99*, number 1579 in LNCS. Springer-Verlag, 1999.

[12] N. S. Bjorner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[13] B. Bollig. Improving the variable ordering of OBDDs is np-complete. *IEEE Transaction on Computers*, 1996.

[14] B. Bollig, M. Lobbing, and I. Wegener. Simulated annealing to improve variable orderings for oBDDs. In *International Workshop on Logic Synthesis*, pages 5b:5.1–5.10, 1995.

[15] Richard John Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge Computer Laboratory, May 1994. Technical Report 337.

[16] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conference*, pages 40–45, 1990.

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

[18] R. E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transaction on Computers*, pages 40:205–213, 1991.

[19] R. E. Bryant and Y. A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Design Automation Conference*, pages 535–541, 1995.

[20] R. E. Bryant, S. German, and M. N. Velve. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Computer-Aided Verification*, LNCS, pages 470–482, 1999.

[21] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification*, volume 818 of *LNCS*, pages 68–80, 1994.

[22] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98:142–170, 1992.

[23] P. Chauhan, E. Clarke, Y. Lu, and D. Wang. Verifying IP-core based System-On-Chip design. In *IEEE ASIC*, September 1999.

[24] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Software Tools for Technology Transfer*, 1998.

[25] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, volume 1855 of *LNCS*, pages 154–169, 2000.

[26] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Publishers, 1999.

[27] E. Clarke, S. Jha, Y. Lu, and D. Wang. Abstract BDDs: a technique for using abstraction in model checking. In *Correct Hardware Design and Verification Methods*, volume 1703 of *LNCS*, pages 172–186, 1999.

[28] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, LNCS, 1981.

[29] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent system using temporal logic. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, January 1983.

[30] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams - overcoming the limitations of MTBDDs and BMDs. In *International Conference of Computer-Aided Design*, pages 159–163, 1995.

[31] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5):1512–1542, September 1994.

[32] E. M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Design Automation Conference*, pages 54–60, 1993.

[33] M. A. Colón and T. E. Uribe. Generating finite-state abstraction of reactive systems using decision procedures. In *Computer-Aided Verification*, pages 293–304, 1998.

[34] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *ACM Symposium of Programming Language*, pages 238–252, 1977.

[35] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and System (TOPLAS)*, 19(2), 1997.

[36] D. R. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of ctl. In *Computer-Aided Verification*, pages 479–490, 1993.

[37] D. R. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems : Abstractions preserving ∀ctl*, ∃ctl*, ctl*. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*, pages 573–592, 1997.

[38] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Computer-Aided Verification*, volume 1633 of *LNCS*, pages 160–171. Springer Verlag, July 1999.

[39] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939 of *Lecture Notes*

*in Computer Science*, pages 54–69, Liege, Belgium, July 1995. Springer Verlag.

[40] R. Drechsler et al. Efficient representation and manipulation of switching functions based on Ordered Kronecker Functional Decision Diagrams. In *Design Automation Conference*, 1994.

[41] R. Drechsler et al. A genetic algorithm for variable ordering of OBDDs. In *IEEE Proceedings of Computer Digital Techniques*, 1996.

[42] E.A. Emerson. Temporal and modal logic. In *Handbook of Theor.Comp.Science. Vol. B.*, pages 995–1072. Elsevier, 1990.

[43] E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–130, 1996.

[44] E.A. Emerson and R.J. Trefler. From asymmetry to full symmetry: new techniques for symmetry reduction in model checking. In *Correct Hardware Design and Verification Methods*, volume 1703 of *LNCS*, pages 142–156, 1999.

[45] M. Fujita et al. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *International Conference of Computer-Aided Design*, 1988.

[46] M. Fujita et al. On variable ordering of Binary Decision Diagrams for the application of multi-level logic synthesis. In *European Design Automation Conference*, 1991.

[47] D.A. Fura, P.J. Windley, and A.K. Somani. Abstraction techniques for modeling real-world interface chips. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving*

*and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 267–281, Vancouver, Canada, August 1993. University of British Columbia, Springer Verlag, published 1994.

[48] M. R. Garey and D. S. Johnson. *Computers and interactability: a guide to the theory of NP-Completeness*. W. H. Freeman And Company, 1979.

[49] Ian Gent and Toby Walsh. The sat phase transition. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 105–109, 1994.

[50] J. Gergov et al. Efficient boolean manipulation with OBDD's can be extended to FBDD's. In *IEEE Transaction of Computers*, 1994.

[51] P. Godefroid, D. Peled, and M. Staskauskas. Using partial order methods in the formal verification of industrial concurrent programs. In *ISSTA'96 International Symposium on Software Testing and Analysis*, pages 261–269, 1996.

[52] Shankar G. Govindaraju and David L. Dill. Verification by approximate forward and backward reachability. In *Proceedings of International Conference on Computer-Aided Design*, November 1998.

[53] S. Graf. Verification of distributed cache memory by using abstractions. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 207–219, Standford, California, USA, June 1994. Springer Verlag.

[54] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification*, volume 1254 of *LNCS*, pages 72–83, June 1997.

[55] P.-H. Ho, A. J. Isles, and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *International Conference of Computer-Aided Design*, pages 529–536, 1998.

[56] R. Hojati and R. K. Brayton. Automatic datapath abstraction in hardware systems. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 98–113, Liege, Belgium, July 1995. Springer Verlag.

[57] G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic*. Methuen, London, 1977.

[58] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, pages 41–76, 1996.

[59] N. Ishiura et al. Minimization of Binary Decision Diagrams based on exchange of variables. In *Design Automation Conference*, 1991.

[60] J. Jain, W. Adams, and M. Fujita. Sampling schemes for computing OBDD variable orderings. In *International Conference of Computer-Aided Design*, 1998.

[61] J. Jain et al. Indexed bdds: Algorithmic advances in techniques to represent and verify boolean functions. *IEEE Transactions on Computers*, 46(11):1230–1245, 1997.

[62] K. Jensen. Condensed state spaces for symmetrical colored petri nets. *Formal Methods in System Design*, 9(1/2):7–40, 1996.

[63] S.-W. Jeong et al. Structural BDDs: Trading canonicity for structure in verification algorithms. In *International Conference of Computer-Aided Design*, 1991.

[64] R. B. Jones, J. U. Skakkebak, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *Formal Methods in Computer-Aided Design*, pages 2–17, 1998.

[65] U. Kebschull et al. Multilevel logic synthesis based on Functional Decision Diagrams. In *European Design Automation Conference*, 1992.

[66] S. Kimura. Residue BDD and its application to the verification of arithmetic circuits. In *Design Automation Conference*, 1995.

[67] R. P. Kurshan. Analysis of discrete event coordination. In *Proceedings of the REX workshop on stepwise refinement of distributed systems, models, formalisms, correctness*, volume 430 of *LNCS*, 1989.

[68] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[69] Y. Lakhnech. personal communication. 2000.

[70] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *International Conference of Computer-Aided Design*, pages 76–81, 1996.

[71] D. Lesens and H. Sadi. Automatic verification of parameterized networks of processes by abstraction. In *International Workshop on Verification of Infinite State Systems (INFINITY)*, Bologna, July 1997.

[72] J. Lind-Nielsen and H. R. Andersen. Stepwise CTL model checking of state/event systems. In *Computer-Aided Verification*, volume 1633 of *LNCS*, pages 316–327. Springer Verlag, 1999.

[73] J. Lind-Nielsen, H. R. Andersen, and H. Hulgaard. Verification of large state/event systems using compositionality and dependency analysis. In *FORM*, 1998.

[74] C. Loiseaux et al. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, pages 1–36, 1995.

[75] D. E. Long. *Model Checking, Abstraction and Compositional Verification*. PhD thesis, School of Computer Science, Carnegie Mellon University, July 1993. CMU-CS-93-178.

[76] S. Malik et al. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference of Computer-Aided Design*, 1988.

[77] Z. Manna et al. Abstraction and modular verification of infinit-state reactive systems. In *Requirements Targeting Software and Systems Engineering (RTSE)*, 1998.

[78] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[79] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods*, September 1999.

[80] C. Meinel, K. Schwettmann, and A. Slobodov. Application driven variable reordering and an example implementation in reachability analysis. In *ASP-DAC*, 1999.

[81] C. Meinel and A. Slobodov. Sample method for minimization of obdds. In *International Workshop of Logic Synthesis*, pages 311–316, 1998.

[82] C. Meinel and C. Stangier. Speeding up symbolic model checking by accelerating dynamic variable reordering. In *10th ACM Great Lake Symposium on VLSI*, 2000.

[83] David G. Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of sat problems. In *Proceedings of Tenth National Conference on Artificial Intelligence*, pages 459–465, 1992.

[84] S. Panda and F. Somenzi. Symmetry detection and dynamic variable ordering of decision diagrams. In *International Conference of Computer-Aided Design*, 1994.

[85] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *International Conference of Computer-Aided Design*, 1995.

[86] A. Pardo. *Automatic Abstraction Techniques for Formal Verification of Digital Systems*. PhD thesis, University of Colorado at Boulder, Dept. of Computer Science, August 1997.

[87] A. Pardo and G.D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference*, pages 457–462, 1998.

[88] PCI SIG Group. *PCI Local Bus Specification*, June 1995.

[89] PCI Special Interest Group. *PCI to PCI Bridge Architecture Specification Rev 1.1*, December 1998.

[90] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, 1977.

[91] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[92] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *Computer-Aided Verification*, pages 84–97, 1995.

[93] K. Ravi, A. Pardo, G. D. Hachtel, and F. Somenzi. Modular verification of multipliers. In *Formal Methods in Computer-Aided Design*, pages pp.49 – pp.63, November 1996.

[94] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference of Computer-Aided Design*, 1993.

[95] J. Rushby. Integrated formal verification: using model checking with automated abstraction, invariant generation, and theorem proving. In *Theoretical and practical aspects of SPIN model checking: 5th and 6th international SPIN workshops*, pages 1–11, 1999.

[96] V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–192, 1999.

[97] H. Saidi and N. Shankar. Abstract and model checking while you prove. In *Computer-Aided Verification*, number 1633 in LNCS, pages 443–454, July 1999.

[98] Bart Selman, Hector Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[99] J. Sifakis. Property preserving homomorphisms of transition systems. In *4th Workshop on Logics of Programs*, June 1983.

[100] E. Solari and G. Willse. *PCI Hardware and Software - Architecture and Design*. Annabooks, 1998.

[101] F. Somenzi. CUDD: CU decision diagram package. Technical report, University of Colorado at Boulder, 1997.

[102] K. Takayama, T. Satoh, T. Nakata, and F. Hirose. An approach to verify a large scale system-on-chip using symbolic model checking. In *International Conference of Computer Design*, pages 308–313, 1998.

[103] M. N. Velve and R. E. Bryant. Exploiting positive equality and partial non-consistency in the formal verification of pipelined microprocessors. In *Design Automation Conference*, pages 397–401, 1999.

[104] M. N. Velve and R. E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *Design Automation Conference*, pages 112–117, 2000.

[105] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the 1989 International*

*Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, 1989.

[106] B. Yang et al. A performance study of BDD-based model checking. In *Formal Methods in Computer-Aided Design*, volume 1522 of *LNCS*. Springer Verlag, 1998.