SAT based Abstraction Refinement for Hardware Verification

Dong Wang
May 2003

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

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

This research is sponsored by the Gigascale Silicon Research Center (GSRC), the National Science Foundation (NSF) under Grant No. CCR-9803774. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of GSRC, NSF, or the United States Government.

Abstract

Model checking is a widely used automatic formal verification technique. Despite the recent advances in model checking technology, its application is still limited by the state explosion problem. For model checking large real world systems, abstraction is essential. This thesis investigates abstraction techniques for the efficient verification of hardware designs with thousands of registers.

A technique, called *SAT conflict dependency analysis*, is developed and used to derive several efficient abstraction algorithms. If a CNF formula is unsatisfiable, this technique can extract a proof of unsatisfiability by analyzing conflict clauses and conflict graphs generated by the SAT procedure.

In this thesis, we propose two new algorithms to improve the efficiency of traditional *localization reduction* based methods. The first algorithm combines multiple verification engines including BDD, ATPG, SAT, and 3-valued simulation for generating abstract counterexamples and for refinement. When the SAT solver determines that there are no concrete counterexamples corresponding to the abstract counterexample, we generate an unsatisfiability proof using the SAT conflict dependency analysis. The second algorithm identifies a set of registers for refinement based on the extracted proof.

Existing *predicate abstraction* techniques are designed for verifying infinite state systems. They become inefficient when applied to the verification of large scale hardware designs. We improve the existing predicate abstraction techniques in several directions. First, computing an abstract model involves many validity checks. A pruning technique is introduced, to avoid the validity checks that are guaranteed to fail. Second, the abstract model is refined by adding compact predicates and general transition constraints identified by unsatisfiability proofs. Third, existing refinement algorithms can add unnecessary predicates, called redundant predicates. We propose algorithms to identify and remove the redundant predicates. Fourth, to exploit high level information from Verilog designs, a method is developed to extract relevant branch conditions that can be used as predicates during refinement. Finally, to improve predicate abstraction further, we combine techniques from localization reduction into the abstraction process.

The abstraction refinement algorithms presented in this thesis have been successfully applied to the verification of industrial hardware designs with up to six thousand registers and 250 thousand gates.

Acknowledgements

I am very fortunate to have Ed Clarke as my adviser during my graduate study. Ed has devoted much effort to teach me how to do research and how to present results effectively. Without his encouragement and invaluable guidance, this thesis would not have been possible. Thanks to all my committee members for their advice and careful reading of my thesis. Randy has given suggestions to extend this work. Don's insight on hardware description languages expands my horizon on verification. Orna has been working with me extensively during her visits to CMU. She has helped me to formalize my ideas and extends them in various directions.

Many thanks to members in our research group. Sergey has always been helpful. I benefit a lot from inspiring discussions with Yuan. His encouragement helps me to get started with research. Helmut, Pankaj and Murali are my research collaborators. Ofer has provided benchmarks used in this thesis. Finally, it has been a great pleasure to work with all the other group members: Will Marrero, Marius Minea, Armin Biere, Yunshan Zhu, Sagar Chaki, Flavio Lerda, Anubhav Gupta, Alex Groce, Nishant Sinha, Daniel Kroening, Joel Ouakinine, Michael Theobald, Karen Yorav and everyone else.

I spent summer 2000 in Synopsys woking with Pei-Hsin Ho, Jiang Long, James Kukula etc. The experience was incredible. The work there becomes an important part of my thesis.

I thank people in ECE and CS facilities for providing an efficient working environment. I would like to acknowledge the financial support of GSRC and NSF. Their support has made this work possible.

I am very grateful to my parents, my brother and my sister. They are always there to support me. And Finally, most of my gratitude goes to my dear wife, Zhuo, for her patience, encouragement and faith in what I am doing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Model checking is an automatic exhaustive search method for the formal verification of finite state systems. As hardware designs are becoming more and more complex, traditional simulation based methodology has been proven insufficient to find subtle design errors. For many hardware designs, the application of model checking is absolutely essential. For example, bugs in high volume electronic products, such as microprocessors, could cause recall of the chips and incur crippling costs to the manufacturers. Furthermore, design errors in mission critical or safety critical systems may cause catastrophic consequences. One major obstacle for the adoption of model checking into the mainstream design flow is the state explosion problem. This thesis investigates abstraction techniques to alleviate this problem, thus enabling the successful application of model checking to verify large scale hardware designs.

1.1 Background

Model checking [25] as introduced by Clarke and Emerson has three basic elements:

- A *Kripke structure* to model the finite state system under verification,
- A formula in computation tree logic (CTL), which belongs to the temporal logic [63] introduced by Pnueli, to specify the property, and
- An efficient model checking algorithm that for each state determines the truth value of subformulas of the given CTL formula.

Since the system states are explicitly manipulated in the original model checking algorithm, only relatively small designs can be verified. Over the past several years, considerable research has been done to improve the basic model checking algorithm of [25]. Symbolic model checking, bounded model checking (BMC), compositional reasoning and abstraction are some of the major techniques to enable model checking of large systems.

Symbolic model checking based on BDDs was introduced by McMillan [50] as a viable solution to alleviate the state explosion problem. In this approach, sets of states and relations are all encoded using ordered binary decision diagrams (OBDD) [14]. Much larger systems [16] have been verified using this method compared to explicit state model checkers. As part of his Ph.D. thesis, McMillan created a symbolic model checker, the *SMV* system. In SMV, the system under verification is described using an SMV program and the property is described using a CTL formula. SMV encodes the system and performs the fixpoint based model checking algorithms only using OBDDs. If the formula does not hold on the model, SMV usually produces a counterexample which is a witness for the failure[1].

A symbolic bounded model checking (BMC) algorithm based on Boolean satisfiability (SAT) solvers was introduced by Biere, Cimatti, Clarke and Zhu in [10]. In BMC, given a linear time temporal logic (LTL) formula, counterexamples of increasing lengths are searched via a reduction to Boolean satisfiability problems. If one of the SAT instances is satisfiable, a counterexample has been found. Recent advances in SAT technology [56, 68, 76] has greatly increased the size of the systems that can be handled by BMC, compared to BDD based model checkers. The effectiveness of BMC has been demonstrated in many verification problems in industry [11,13,26]. However, in order to show the correctness of the LTL formula, a large bound may be necessary. A number of approaches [2,12,37,53,67,73] have been investigated to use SAT solvers for unbounded model checking.

Compositional reasoning [1, 4, 36, 39, 51, 52, 62] is used to reduce the verification of a large system to a number of smaller verification problems. The correctness of the overall system is then established by composing the proofs of correctness of various parts. In this approach, properties of each part are verified by making assumptions on the behavior of other parts. For a part, other parts act as an environment. These assumptions must be proved later when the correctness of other parts is proved. In practice, a complex hardware system is broken into smaller blocks based on the design modularity.

[1]Counterexamples are produced only for universal CTL formulas which have either a path or a loop counterexample. Other counterexamples are not produced, only their falsehood is indicated.

For the verification of each block, the assumptions about the environment are specified as input constraints. Over-constraining the inputs will result in false confidence in the correctness of the design. On the other hand, under-constraining the inputs leads to false errors. The difficulty in generating the exact input constraints is a major obstacle to the application of compositional reasoning in industry.

Abstraction is an effective method to alleviate the state explosion problem. There are many abstraction techniques, including the localization reduction [9,18,22,33,43,72], the homomorphic abstraction [21,23,64], the abstraction without explicit abstraction function [60], abstract interpretation [46], the free and constrained abstractions [27], predicate abstraction [6, 65, 66], etc. All abstraction techniques compute abstract models of the given concrete system by leaving out "irrelevant" details, thus model checking the abstract models is considerably simpler than directly applying model checking to the concrete system. For an abstraction technique to be effective, it is important to come up with the right kind of abstract models to preserve the relevant behavior for the property. Usually, the initial abstraction is too coarse. It needs to be successively refined to gradually add more behavior to determine the result of the verification. It is desirable to keep the abstract model as small as possible, while still being sufficient for the verification. The existing abstraction techniques do not consider refinement at all, or the refinement process is computationally expensive or ineffective. Therefore, the major challenge is to automate the refinement process to find the small and sufficient abstract models efficiently.

1.2 Scope of This Thesis

This thesis investigates new abstraction techniques for the efficient verification of large scale hardware designs. Two abstraction methods, the localization reduction and the predicate abstraction, have been enhanced to solve the following problems:

1. In localization reduction, abstract models are constructed by retaining certain parts of the concrete model. Thus the size of the abstract models can be large, which could make model checking even the abstract models computationally expensive.

2. In localization reduction, for hardware designs with thousands of registers, identifying a small set of registers to build the abstract model required for the verification of the given property is difficult. Also, existing techniques invalidate one abstract counterexample at a time for refinement.

3. Existing predicate abstraction techniques are suitable for the verification of infinite state systems. However, they are inefficient when applied to the verification of large scale hardware systems. In particular, the algorithm to build the abstract model, the algorithm to refine the abstract transition relation and the algorithm to compute new predicates are not effective for hardware verification.

4. Predicate abstraction may perform badly for control intensive systems, because simulating the control structure may require a large number of predicates.

5. Although a typical design flow starts at register transfer level (RTL), existing model checking engines and verification tools use the gate level representation of the design under verification. High level RTL designs are synthesized to lower gate level designs before a verification tool can read and encode them. High level design information, e.g. predicates in Verilog descriptions, can be crucial for the success of the verification.

6. Counterexample guided abstraction refinement may add redundant predicates that are not necessary for the verification of the given property. This unnecessarily increases the size of the abstract model. There is no automatic way to identify and remove those redundant predicates.

The goal of this thesis is to address these problems. The principal contributions of this work are detailed below:

SAT conflict dependency analysis. Our abstraction refinement algorithms are based on SAT procedures. These algorithms rely on the identification of unsatisfiability proofs of SAT formulas. A technique, called SAT conflict dependency analysis, is developed to extract a small unsatisfiability proof of the given SAT formula. Using the unsatisfiability proof, a subformula of an unsatisfiable SAT formula can be shown to be unsatisfiable. Based on SAT conflict dependency analysis, an incremental SAT solver is built on top of the modern SAT solver zChaff [56], which can considerably speed up solving a set of related SAT problems whether they are satisfiable or not.

Localization reduction based on multiple verification engines. Localization reduction overapproximates the given concrete model by keeping a set of important registers (visible registers) and hiding the rest (invisible registers). We enhance traditional localization reduction for the verification of hardware designs with thousands of registers. A hybrid BDD/ATPG algorithm is developed to compute efficiently abstract counterexamples for abstract models with large number of inputs. A 3-valued simulator and a SAT solver are used to identify a minimal set of registers that cause the given abstract counterexample to fail on the concrete model.

Localization reduction based on unsatisfiability proofs. A new localization reduction algorithm is developed, where the refinement only adds those invisible registers that appear in the unsatisfiability proof generated when the abstract counterexample is proved to be unsatisfiable on the concrete model. Furthermore, we propose two new algorithms to generalize counterexamples based localization reduction. The first algorithm generates better initial abstractions by the use of BMC and the unsatisfiability proofs of the corresponding SAT formulas. The second algorithm is used to invalidate multiple abstract counterexamples at once.

SAT based predicate abstraction for RTL Verilog design verification. The construction of an abstract model using predicate abstraction involves potentially exponential number of validity checks. Each of these checks requires one call to the SAT solver. We developed a pruning technique to reduce the number of calls to the SAT solver. To eliminate a spurious abstract counterexample, two efficient SAT based refinement algorithms are developed. The first algorithm requires only one call to the SAT solver, while the number of calls in the existing algorithm [28] is twice the number of predicates. Our second refinement algorithm computes new predicates that are compact, while the existing algorithm [65] does not consider the size of the new predicates. Furthermore, we developed a practical method to exploit high level information in predicate abstraction. This method extracts relevant branch conditions in RTL Verilog designs before verification starts. To use these branch conditions in predicate abstraction, a lazy refinement algorithm is developed. This algorithm identifies a subset of the branch conditions that can invalidate a spurious counterexample without constructing the full refined abstract model.

Combine localization reduction with predicate abstraction. For control variables that determine the behavior of the concrete system, the number of predicates required to simulate their behavior may be much larger than the number of control variables. We develope a clustering based heuristic to identify when such a blow up of the abstract model is likely to occur for predicate abstraction. Then a modified localization reduction algorithm is used to include these variables into the abstract model. Furthermore, it is usually the case that different predicates are not independent. Efficient algorithms are designed to compute constraints between predicates. The computed constraints are added as invariants to the abstract model to make it more accurate.

Removing redundant predicates. Existing predicate abstraction algorithms use counterexamples to guide the computation of new predicates. For the verification of the given property, it is possible for the refinement algorithm to include unnecessary predicates, called redundant predicates. We have developed two criteria to identify when a predicate is redundant based on the concept of *replacement functions*. We also show how to remove the redundant predicates efficiently, once they are identified.

I believe that the novel ideas presented significantly advance the state of the art in hardware verification. We performed a large number of experiments on industrial benchmarks with thousands of registers and greatly improved upon the results of the existing verification techniques.

1.3 Related Work

In this section, we briefly review some of the related work.

1.3.1 SAT Unsatisfiability Proofs

Extracting unsatisfiability proofs is also studied by Zhang and Malik in [75] and by McMillan and Amla in [55]. Their approaches to extract unsatisfiability proofs are similar to the one presented in this thesis, except that they use resolution rather than the boolean constraint propagation method to represent the reasoning in a conflict graph.

In [75], experimental study is performed to measure the quality of the extracted proofs. For their examples, only 19% to 90% of the generated conflict clauses during SAT search are actually needed in the proofs. For an unsatisfiable CNF formula, they also show through experiments that the set of clauses in the unsatisfiability proof is not the minimal unsatisfiable subformula, confirming the claim in this thesis. Compared with our work, the extracted unsatisfiability proofs are used for different purposes. We use unsatisfiability proofs to perform counterexample guided abstraction refinement; while in [75], the unsatisfiability proofs are verified to check the correctness of the SAT solver. McMillan and Amla use unsatisfiability proofs to perform localization reduction without counterexamples. We will discuss their algorithm in the following subsection.

Incremental SAT is independently studied by Kim, Whittemore and Sakallah in [42]. They show how an incremental SAT solver can be used to solve a set of related SAT problems, where constraints are added in last-in-first-out order. Similar to our dependency analysis, their algorithm to enable the reuse of conflict clauses maintains the relationship between a conflict clause and the clauses that are responsible for it.

1.3.2 Localization Reduction and Counterexample Guided Refinement

In [43,64], Kurshan proposed the high-level strategy called localization reduction for the language containment problem between a system of L-processes and a specification of the system in terms of L-automata. The abstract models are subsets of the L-processes. Refinement is based on adding L-processes to invalidate the abstract counterexamples, which is guided by the dependency graph among L-processes. However, the description of the algorithm does not provide enough details to implement a practical tool.

Balarin et al. [5] reported a similar iterative algorithm for checking language emptiness of networks of communicating automata. The abstract models are subsets of the communicating automata. Refinement is based on adding some extra communicating automata to the abstract model. The choice is based on the degree of common support between the current abstract model and the automata that have not been included in the abstract model. The verification result of a collection of dining philosophers using BDD-based image computation is reported.

Lu [48] developed a counterexample guided abstraction refinement framework. Lu was the first to propose refinement based on separation of *deadend* and *bad* states. Our refinement algorithm is also based on this concept. However, there are several differences. The biggest bottleneck in his method is the use of BDD based image computations *on concrete systems* for validating counterexamples. We use symbolic simulation based on SAT to accomplish this task. His method to separate deadend and bad states is based on splitting the variable domains, while our methods either hide irrelevant parts of the design or introduce new predicates.

In [22], Clarke et al. proposed localization reduction algorithms based on separation of deadend and bad states using integer linear programming (ILP) and machine learning techniques. They sample the deadend and bad states and produce optimal separating variables for the samples. This process is repeated till the separating variables are sufficient to separate deadend and bad states.

Subsequently, Chauhan et al. proposed a SAT conflict analysis based heuristic score algorithm for refinement in [18]. This algorithm analyzes the structure of SAT search to identify important registers. The algorithm is computationally inexpensive and does not need multiple SAT checks for refinement. We also introduced SAT conflict dependency analysis to extract unsatisfiability proofs in [18]. The set of registers identified by heuristic score method for refinement is usually larger than that identified by SAT unsatisfiability proof based methods.

In [55], McMillan and Amla proposes a new localization reduction algorithm that does not use abstract counterexamples to perform refinement. Instead, in each iteration of the abstraction refinement procedure, SAT based bounded model checking with increasing bounds is performed on the concrete model. If there is no concrete counterexample of a given length, the registers in the extracted unsatisfiability proofs are used to construct an abstract model. Then, BDD based model checking is used to verify the property on the abstract model. If the property is false on the abstract model, the above procedure is repeated. Their algorithm is similar to the proof based method to extract a set of important registers from bounded model checking presented in Section 4.4.4. However, there are important differences. We use the method based on BMC only to generate initial abstraction. Moreover, we are not required to use all the registers in the unsatisfiability proofs, since the proofs are not minimal and there could be too many registers in the proofs.

1.3.3 Predicate Abstraction

Predicate abstraction was introduced by Graf and Saidi in [65]. They used PVS theorem prover to perform on the fly Overapproximate reachability analysis of infinite state systems. In their approach, the abstract state space is represented as monomials over predicates. No abstract model is explicitly built. In [66], Saidi and Shankar introduced an algorithm to compute abstractions for infinite state systems that preserve all $\mu-$calculus formulas. Their algorithm does not introduce any approximations in the abstract model, however, it requires an exponential number of validity checks. They present a simplistic refinement process to remove the nondeterministic behaviors introduced by predicate abstraction.

An algorithm to make the abstract model more accurate given a fixed set of predicates is presented in [28]. To speed up the abstraction process, they introduce approximations in the abstract model. This results in spurious transitions in the abstract model. To remove a spurious transition, their algorithm requires $2m$ number of calls to a theorem prover, where m is the number of predicates. Our algorithm is more efficient in that no additional calls to a SAT solver are required. Note that, in general, their algorithm can come up with a more general constraint than ours. However, we can get the same constraints, probably using much less time, by combining both algorithms together. Furthermore, the work in [28] does not consider the problem of introducing new predicates to refine the abstract model.

Exploiting high level hardware description language features for abstraction has been investigated in [23]. They extract conditions of **case** statements in the SMV language in order to build the initial abstraction. The extraction method in [23] requires modifying the *source code* of an existing translator of SMV language. We transform the given Verilog design to an equivalent design where the predicates are uniquely named. The modified design can be processed by commercial synthesis tools to generate verification models where the predicates are preserved. In [23], the extracted conditions are used only for the initial abstraction; while we use the predicates for both initial abstraction and refinement.

Lazy abstraction for the verification of C programs has been investigated in [38]. The goals of their algorithm and ours are different. In [38], the construction of the abstract model and abstract model checking are performed only from the state where the spurious abstract counterexample fails on the concrete system. While our lazy refinement algorithm builds the refined abstract model only to the point where the counterexample is invalidated.

Some researchers have considered combining unabstracted control variables (visible variables) with predicate abstraction [57], but their methods are not automatic. Using the correlations between all predicates to constrain the abstract model has been investigated in [6]. The correlations are computed using a general theorem prover. We first partition the set of predicates into clusters based on the sharing of support sets, then correlations are computed for each cluster separately. Although our result is more approximate, the complexity of our algorithm is much less sensitive to the total number of predicates. We also give a BDD-based algorithm to compute the correlations between predicates. We also given an algorithm to compute the correlations between unabstracted control variables and predicates. As far as we know, no one else has considered these kind of correlations before.

Similar to our algorithms for removing redundant predicates, in [6] a technique called *strengthening* is proposed. To build the abstract model, the weakest precondition is converted to an expression over the set of predicates in the abstraction. Thus, strengthening is somewhat similar to the concept of *replacement functions* in this thesis. However, in [6], the result of the strengthening is over all the predicates, while the replacement functions used here are defined over a subset of the predicates. Finally, the two transformations have different purposes. Strengthening is only used to build an abstract model; while our transformation is used to remove redundant predicates and thus reduce the complexity of the abstract model.

1.3.4 Other Abstraction Techniques

Rather than building abstract models explicitly and relying on counterexamples to guide the refinement, Pardo and Hachtel [60] used BDD subsetting to perform on-the-fly abstraction and refinement. Based on the polarity of a CTL subformula, under or over approximation is used. In our experience, subsetting-based abstraction methods are very unpredictable and too drastic to prove properties. The scalability problem of BDD-based methods also makes finding real counterexamples on original designs with thousands of registers almost impossible.

In [35], Govindaraju and Dill proposed an abstraction refinement algorithm for verifying safety properties. The abstract models are collections of state machines that form an overlapping partition of the original design. Post-image and pre-image computation methods are used to prove the property or generate an abstract counterexample on the partitioned design. Refinement is based on enlarging individual state machines in the overlapping partition of the original design, guided by heuristics based on the Hamming distance. An experiment on the verification of a PCI chip with 429 latches is reported. We believe that this method also suffers from the scalability issue of BDD-based methods, and it will have difficulties in handling big designs even when they are partitioned.

Chapter 2

Existential Abstraction

In this chapter we review the relevant theory of existential abstraction introduced by Clarke, Grumberg and Long in [21] and Loiseaux et al. in [46]. Using this theory we describe the predicate abstraction framework of Saidi and Shankar [66] and the localization reduction [43]. To handle spurious abstract counterexamples, the abstraction refinement framework is introduced.

2.1 Notation

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of variables, where each variable v_i has a domain D_{v_i}. Let c be a function which maps each variable $v_i \in V$ to a value in its domain D_{v_i}. If $V_1 \subseteq V$, the *projection* of c over V_1, denoted by $\mathit{proj}[V_1](c)$, is a function defined over V_1 that is consistent with c over V_1.

Let S_1 and S_2 be sets of states, and let f be a function mapping the powerset of S_1 to the powerset of S_2, i.e., $f : 2^{S_1} \rightarrow 2^{S_2}$. The *dual of the function* f is defined to be

$$\widetilde{f}(X) = \overline{f(\overline{X})},$$

where the overbar indicates complementation in the appropriate set of states.

Let ρ be a relation from S_1 to S_2, and let A be a subset of S_2, then the function $pre[\rho](A)$ gives the *preimage* of A under the relation ρ. Formally,

$$pre[\rho](A) = \{s_1 \in S_1 \ \mid \ \exists s_2 \in A.\ \rho(s_1, s_2)\}.$$

Similarly, let B be a subset of S_1, then the function $post[\rho](B)$ gives the *postimage* of B under the relation ρ. More formally,

$$post[\rho](B) = \{s_2 \in S_2 \ \mid \ \exists s_1 \in B.\ \rho(s_1, s_2)\}$$

Lemma 2.1.1 *[46] If relation ρ is a total function on S_1, then $\widetilde{pre}[\rho]$ is the same as $pre[\rho]$*

Proof: We prove it by showing that $\forall S \in 2^{S_2}\ pre[\rho](S) = \widetilde{pre}[\rho](S) = \overline{pre[\rho](\overline{S})}$. First, if $x \in pre[\rho](S)$, then there exists $s \in S$ such that $\rho(x, s)$ holds. Hence there does not exists $y \in \overline{S}$ such that $\rho(x, y)$ holds (since ρ is a function). Thus $\neg(x \in pre[\rho](\overline{S}))$. So $x \in \overline{pre[\rho](\overline{S})}$.

Next, suppose $x \in \overline{pre[\rho](\overline{S})}$, then $\neg(x \in pre[\rho](\overline{S}))$. Now ρ is total so $\rho(x, s)$ holds for some s. Since $\neg(x \in pre[\rho](\overline{S})$, it follows that $\neg(s \in \overline{S})$. Thus $s \in S$ and $x \in pre[\rho](S)$. ∎

We will be reasoning about a concrete state machine and an abstraction of that machine. To establ a relationship between the set of concrete states S_1 and the set of abstract states S_2 we will use the concept of a *Galois connection.*

Definition 2.1.1 Let Id_S denotes the identity function on the powerset of S. A *Galois connection* between 2^{S_1} and 2^{S_2} is a pair of monotonic functions (α, γ), where $\alpha : 2^{S_1} \rightarrow 2^{S_2}$ and $\gamma : 2^{S_2} \rightarrow 2^{S_1}$, such that $Id_{S_1} \subseteq \gamma \circ \alpha$ and $\alpha \circ \gamma \subseteq Id_{S_2}$.

The following duality property of Galois connections is well known [46].

Proposition 2.1.1 For any Galois connection (α, γ) from 2^{S_1} to 2^{S_2}, we have,

- $\gamma(Y) = \bigcup\{X \in 2^{S_1} \mid \alpha(X) \subseteq Y\}$,
- $\alpha(X) = \bigcap\{Y \in 2^{S_2} \mid X \subseteq \gamma(Y)\}$.

Note that, given either one of α or γ, the other is uniquely determined. Typically, α and γ are used to define the relationship between the abstract and concrete models. The functions α and γ are often called the *abstraction function* and the *concretization function*, respectively. The Galois connection that we will be using in this paper is described in the following proposition.

Proposition 2.1.2 [46] Given a relation $\rho \subseteq S_1 \times S_2$, the pair $(post[\rho], \widetilde{pre}[\rho])$ is a Galois connection between 2^{S_1} and 2^{S_2}.

We denote this Galois connection by $(\alpha_\rho, \gamma_\rho)$. Sometimes, we write (α, γ) when the relation ρ is clear from the context.

2.2 Existential Abstraction

We model circuits and programs as transition systems. Given a set of atomic propositions, A, let $M = (S, S_0, R, L)$ be a *transition system*, where S is the set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation and $L\ : S \rightarrow 2^A$ is the labeling function from the set S to the powerset of A. Often a state of a system will be described as an assignment of values to the set of state variables $V = \{v_1, v_2, .., v_m\}$. In this case, R will be given as a formula over two copies of the state variables, one representing the *current state* and the other the *next state*. If the set of current state variables is $V = \{v_1, v_2, .., v_m\}$ then the set of next state variables is $V' = \{v'_1, v'_2, .., v'_m\}$. Note that functions that are applicable to unprimed variables will be applicable to the corresponding primed versions too, the only difference is that the result will also be in terms of primed variables.

Definition 2.2.1 Given two transition systems $M = (S, S_0, R, L)$ and $\hat{M} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$, with atomic propositions A and $\hat{A}$ respectively, a relation $\rho \subseteq S \times \hat{S}$, which is total on S, is a *simulation relation* between M and $\hat{M}$ if and only if for all $(s,\ \hat{s}) \in \rho$ the following conditions hold:

- $L(s) \bigcap \hat{A}\ = \hat{L}(\hat{s}) \bigcap A$
- For each state s_1 such that $(s, s_1) \in R$, there exists a state $\hat{s_1} \in \hat{S}$ with the property that $(\hat{s}, \hat{s_1}) \in \hat{R}$ and $(s_1, \hat{s_1}) \in \rho$.

We say that $\hat{M}$ *simulates* M through the simulation relation ρ, denoted by $M \preceq_\rho \hat{M}$, if for every initial state s_0 in M there is an initial state $\hat{s_0}$ in $\hat{M}$ such that $(s_0, \hat{s_0}) \in \rho$. We say that ρ is a *bisimulation relation* between M and $\hat{M}$ if $M \preceq_\rho \hat{M}$ and $\hat{M} \preceq_{\rho^{-1}} M$. If there is a bisimulation relation between M and $\hat{M}$ then we say that M and $\hat{M}$ are *bisimilar*, and we denote this by $M \equiv_{bis} \hat{M}$.

Given a transition system M and a CTL^* formula f on the atomic propositions A associated with M, the satisfaction relation $\models$ is defined in the standard fashion (see [24]). The following is a well-known theorem relating the formulas satisfied by two transitions systems where one simulates the other (see [21, 24]).

Theorem 2.2.1 (Preservation of ACTL* [24])
Let $M = (S, S_0, R, L)$ and $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$ be two transition systems, with A and $\hat{A}$ as the respective sets of atomic propositions and let $\rho \subseteq S \times \hat{S}$ be a relation such that $M \preceq_\rho \hat{M}$. Then, for any ACTL* formula Φ with atomic propositions in $A \cap \hat{A}$

$$\hat{M} \models \Phi \text{ implies } M \models \Phi.$$

Theorem 2.2.2 (Preservation of CTL* [24])
Let $M = (S, S_0, R, L)$ and $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$ be two transition systems, with A and $\hat{A}$ as the respective sets of atomic propositions and let $\rho \subseteq S \times \hat{S}$ be a bisimulation relation between M and $\hat{M}$. Then, for any CTL* formula Φ with atomic propositions in $A \cap \hat{A}$

$$\hat{M} \models \Phi \;\Leftrightarrow\; M \models \Phi.$$

The reader is referred to [46] and [21] for details of the proof. The former paper uses a different notation.

Let $M = (S, S_0, R, L)$ be a concrete transition system over a set of atomic propositions A. Let $\hat{S}$ be a set of abstract states and $\rho \subseteq S \times \hat{S}$ be a total function on S. Further, let ρ and L be such that for any $\hat{s} \in \hat{S}$, all states in $pre[\rho](\hat{s})$ have the same labeling over a subset $\hat{A}$ of A. Then an abstract transition system $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$ over $\hat{A}$ which simulates M can be constructed as follows:

$$\hat{S}_0 = post[\rho](S_0) = \exists s.\ S_0(s) \wedge \rho(s, \hat{s}) \tag{2.1}$$

$$\hat{R}(\hat{s}, \hat{s}') = \exists s\ s'.\ \rho(s, \hat{s}) \wedge \rho(s', \hat{s}') \wedge R(s, s') \tag{2.2}$$

$$\text{for each } \hat{s} \in \hat{S}, \hat{L}(\hat{s}) = \bigcap_{s \in pre[\rho](\hat{s})} (L(s) \cap \hat{A}) \tag{2.3}$$

Proposition 2.2.1 For M and $\hat{M}$ in the above construction $M \preceq_\rho \hat{M}$

In the above construction $\hat{R}$ is defined in terms of the abstract current state $\hat{s}$ and the abstract next state $\hat{s}'$. This construction is from [21], and it is also implicit in the paper by Loiseaux et al. [46]. The idea behind the transition system is as follows: two abstract states are related if there exist two concrete states that they are related to each other under the concrete relation and map to the abstract states under ρ. This kind of abstraction is called *existential abstraction.* The set of initial states in the abstract system are those states of $\hat{S}$ that are related to the initial states of M. Note that for any two states s and $\hat{s}$ related under ρ the property $L(s) \cap \hat{A} = \hat{L}(\hat{s})$ holds.

2.3 Predicate Abstraction

Predicate abstraction [6, 7, 28, 29, 38, 57, 65, 66], can be viewed as a special case of existential abstraction. In predicate abstraction a set of predicates $\{P_1, \ldots, P_k\}$, including those in the property to be verified, are identified from the concrete program. These predicates are defined on the variables of the concrete system. They also serve as the atomic propositions that label the states in the concrete and abstract transition systems. That is, the set of atomic propositions is $A = \{P_1, P_2, .., P_k\}$. A state in the concrete system will be labeled with all the predicates it satisfies. The abstract state space has a boolean variable B_j corresponding to each predicate P_j. So each abstract state is a valuation of these k boolean variables. An abstract state will be labeled with predicate P_j if the corresponding bit B_j is 1 in that state. The predicates are also used to define a total function ρ between the concrete and abstract state spaces. A concrete state s will be related to an abstract state $\hat{s}$ through ρ if and only if the truth value of each predicate on s equals the value of the corresponding boolean variable in the abstract state $\hat{s}$. Formally,

$$\rho(s, \hat{s}) = \bigwedge_{1 \leq j \leq k} P_j(s) \Leftrightarrow B_j(\hat{s}) \tag{2.4}$$

Note that ρ is a total function because each P_j can have one and only one value on a given concrete state and so the abstract state corresponding to the concrete state is unique. Based on Section 2.1, the pair of functions $post[\rho]$ and $\widetilde{pre}[\rho]$ generated from relation ρ forms a Galois connection. We will denote this Galois connection by (α, γ). Note that since ρ is a total function, $\widetilde{pre}[\rho] = pre[\rho]$. The following lemma establishes that the set of concrete states corresponding to a set of abstract states can be computed by simply substituting predicates for the bits in the formula representing the abstract states. In the following, $\hat{Y}[B_i \leftarrow P_i]$ denotes the formula obtained by substituting each boolean variable B_i by the corresponding predicate P_i.

Lemma 2.3.1 *Let ρ be an abstraction function. For a set of abstract states*

$\hat{Y}$, $\gamma_\rho(\hat{Y}) = \hat{Y}[B_i \leftarrow P_i]$

Proof:

$$\begin{array}{lll}
 & \gamma_\rho(\hat{Y}) & \\
= & pre[\rho](\hat{Y}) & \textit{since } \widetilde{pre}[\rho] = pre[\rho] \\
= & \exists \hat{s}.\ \hat{Y}(\hat{s}) \wedge \rho(s, \hat{s}) & \text{definition of } pre[\rho] \\
= & \exists \hat{s}.\ \hat{Y}(\hat{s}) \wedge \bigwedge_{1 \le i \le k} P_i(s) \Leftrightarrow B_i(\hat{s}) & \text{definition of } \rho \\
= & \hat{Y}[B_i \leftarrow P_i] & \text{definition of substitution}
\end{array}$$

∎

Using this ρ and the construction given in Section 2.2, we can build an abstract model which simulates the concrete model. Since $(\alpha_\rho, \gamma_\rho)$ is a Galois connection, for any concrete state X, $\alpha_\rho(X) = \bigwedge\{\hat{f} \mid X \rightarrow \gamma_\rho(\hat{f})\}$, where $\hat{f}$ is an arbitrary formula over the abstract state variables. According to [66], it is enough to only consider all disjunctions over the abstract state variables in calculating abstractions. Thus the set of abstract initial states for predicate abstraction is:

$$\hat{S}_0 = \bigwedge\{\hat{Y_1} \mid \forall V.(S_0 \Rightarrow \gamma(\hat{Y_1}))\} \tag{2.5}$$

In equation (2.5), $\hat{Y_1}$ is an arbitrary disjunction of the literals of the current state variables $\{B_1, B_2, \ldots, B_k\}$. In [66] the abstract transition relation $\hat{R}$ is defined as

$$\bigwedge\{\hat{Y} \rightarrow \hat{Y}' \mid \forall V,\ V'.(R(V, V') \Rightarrow \gamma(\hat{Y} \rightarrow \hat{Y}'))\} \tag{2.6}$$

In equation (2.6), $\hat{Y}$ is an arbitrary conjunction of the literals of the current state variables $\{B_1, B_2, \ldots, B_k\}$ and $\hat{Y}'$ is an arbitrary disjunction of literals of the next state variables $\{B'_1, B'_2, \ldots, B'_k\}$. In the above two equations, V is the set of concrete current state variables and V' is the set of concrete next state variables. The set of concrete initial states S_0 is represented by its characteristic function over V. Similarly, the concrete transition relation R is represented by its characteristic function over $V \cup V'$. Note that $\gamma(\hat{Y_1})$ can be represented by a formula over V, and $\gamma(\hat{Y} \rightarrow \hat{Y}')$ can be represented by a formula over $V \cup V'$. Thus the implications in the equations (2.5) and (2.6) are well formed. It is easy to see that both the set of abstract initial states and the abstract transition relation in the above equations are the most accurate overapproximation of the set of concrete initial states and concrete transition relations respectively. We will show that (2.6) is equivalent to (2.2).

Theorem 2.3.1 Let R be a concrete transition relation, ρ be a simulation relation as in (2.4), then

$$\begin{gathered}\exists s\ s'.\ \rho(s, \hat{s}) \wedge \rho(s', \hat{s}') \wedge R(s, s') = \\ \bigwedge\{\hat{Y} \rightarrow \hat{Y}' \mid (\forall V,\ V'.(R(V, V') \wedge \gamma(\hat{Y})) \rightarrow \gamma(\hat{Y}'))\}\end{gathered}$$

where $\hat{Y}$ is a conjunction over $\{B_1, \ldots, B_k\}$ and $\hat{Y}'$ is a disjunction over $\{B'_1, \ldots, B'_k\}$.

Proof: First we prove that if two states are related under

$$R_1 = \exists s\ s'.\ \rho(s, \hat{s}) \wedge \rho(s', \hat{s}') \wedge R(s, s')$$

then they are related under $R_2 = \bigwedge\{\hat{Y} \to \hat{Y}' \mid \forall V,\ V'.((R(V,V') \wedge \gamma(\hat{Y})) \to \gamma(\hat{Y}'))\}$. Suppose $R_1(\hat{s}, \hat{s}'))$ and let the corresponding concrete states be s and s'. So $\rho(s, \hat{s})$ and $\rho(s', \hat{s}')$ hold. To prove that $\hat{s}$ and $\hat{s}'$ satisfy R_2, we need to show that every implication of the form $\hat{Y} \to \hat{Y}'$, which satisfies $(R \wedge \gamma(\hat{Y})) \to \gamma(\hat{Y}')$, is true for $\hat{s}, \hat{s}'$. Suppose $\hat{Y}(\hat{s})$ is false, then the implication $\hat{Y} \to \hat{Y}'$ is automatically true for $(\hat{s}, \hat{s}')$. Consider the case where $\hat{Y}(\hat{s})$ is true. We know that state s satisfies $\rho(s, \hat{s})$ and since $\gamma = \mathit{pre}[\rho]$, thus $s \in \gamma(\hat{Y})$. Now since $(R \wedge \gamma(\hat{Y})) \to \gamma(\hat{Y}')$, $s \in \gamma(\hat{Y})$ and $R(s, s')$, we also have $s' \in \gamma(\hat{Y}')$. Now we show that $\hat{s}' \in \hat{Y}'$.

$$\begin{array}{lr}
s' \ \in \gamma(\hat{Y}') & \text{just proved} \\
\alpha(s') \ \subseteq \alpha(\gamma(\hat{Y}')) & \text{monotonicity of } \alpha \\
\alpha(\gamma(\hat{Y}')) \subseteq \hat{Y}' & \alpha \circ \gamma \subseteq Id_{\hat{S}} \\
\hat{s}' \in \alpha(s') & \alpha(s') = \mathit{post}[\rho](s') \text{ and } \rho(s', \hat{s}') \\
\hat{s}' \in \hat{Y}' & \text{by the above results.}
\end{array}$$

Therefore, we have shown that every implication $\hat{Y} \to \hat{Y}'$, which satisfies $(R \wedge \gamma(\hat{Y})) \to \gamma(\hat{Y}')$, is true for any pair of states $(\hat{s}, \hat{s}')$ related under R_1.

For the other direction, we need to prove that if two states are related through R_2 then they are related under R_1. Equivalently, we can prove that if two states are not related under R_1 then they cannot be related under R_2 either. Consider two states $\hat{s}$ and $\hat{s}'$ which are not related under R_1. We will show that there exists an implication $\hat{Y} \to \hat{Y}'$ that satisfies $(R \wedge \gamma(\hat{Y})) \to \gamma(\hat{Y}')$ and is false for the pair $(\hat{s}, \hat{s}')$. We define two formulas

$C_{\hat{s}}$ and $C'_{\hat{s}'}$ as follows.

$$C_{\hat{s}} = \bigwedge\{B_i \mid \hat{s}(B_i) = 1\} \wedge \bigwedge\{\neg B_i \mid \hat{s}(B_i) = 0\}$$

$$C'_{\hat{s}'} = \bigwedge\{B'_i \mid \hat{s}'(B'_i) = 1\} \wedge \bigwedge\{\neg B'_i \mid \hat{s}'(B'_i) = 0\}$$

We will show that $C_{\hat{s}} \rightarrow \neg C'_{\hat{s}'}$ is the required implication. Clearly, the implication $C_{\hat{s}} \rightarrow \neg C'_{\hat{s}'}$ is false for the pair $(\hat{s}, \hat{s}')$ because by definition $C_{\hat{s}}(\hat{s})$ is true and $\neg C'_{\hat{s}'}(\hat{s}')$ is false. To complete the proof we just need to show that $(R \wedge \gamma(C_{\hat{s}})) \rightarrow \gamma(\neg C'_{\hat{s}'})$ is true. We first show that $\rho(s, \hat{s}) \Leftrightarrow s \in \gamma(C_{\hat{s}})$.

$$\begin{aligned} & \rho(s, \hat{s}) \Leftrightarrow s \in \mathit{pre}[\rho](\hat{s}) \\ \Leftrightarrow \quad & s \in \gamma(\hat{s}) \quad (\text{ since } \gamma = \mathit{pre}[\rho] \text{ }) \\ \Leftrightarrow \quad & s \in \gamma(C_{\hat{s}}) \quad (\text{ since } C_{\hat{s}} = \{\hat{s}\} \text{ }). \end{aligned}$$

Next we show that $\neg\rho(s', \hat{s}') \Leftrightarrow s' \in \gamma(\neg C'_{\hat{s}'})$.

$$\begin{aligned} & \neg\rho(s', \hat{s}') \Leftrightarrow s' \notin \mathit{pre}[\rho](\hat{s}') \\ \Leftrightarrow \quad & s' \notin \widetilde{\mathit{pre}}[\rho](\hat{s}') \text{ (since } \rho \text{ is a total function)} \\ \Leftrightarrow \quad & s' \in \mathit{pre}[\rho](\overline{\hat{s}'}) \\ \Leftrightarrow \quad & s' \in \gamma(\overline{\hat{s}'}) \\ \Leftrightarrow \quad & s' \in \gamma(\neg C'_{\hat{s}'}). \end{aligned}$$

Finally,

$$\begin{aligned}
& (\hat{s}, \hat{s}') \notin R_1 \\
\Leftrightarrow\ & \forall s, s'.\ \neg(\rho(s, \hat{s}) \wedge \rho(s', \hat{s}') \wedge\ R(s, s')) \\
\Leftrightarrow\ & \forall s, s'.\ (R(s, s') \wedge \rho(s, \hat{s})) \Rightarrow \neg\rho(s', \hat{s}') \\
\Leftrightarrow\ & \forall s, s'.\ (R(s, s') \wedge \gamma(C_{\hat{s}})(s)) \Rightarrow \gamma(\neg C'_{\hat{s}'})(s')
\end{aligned}$$

Thus $(\hat{s}, \hat{s}') \notin R_2$. which is the required result. ∎

There are several reasons to prefer (2.6) over (2.2) for computing the abstract transition relation. Traditionally (2.2) is computed using BDDs, but this method is not feasible for the large systems considered in this work. Alternatively, we could formulate (2.2) as a SAT problem. Computing the abstract transition relation would then require enumerating all possible satisfying assignments to the SAT formula. Furthermore, it is not easy to get an over-approximation using the SAT formulation of (2.2). However, in formula (2.6), an implication of the form $\hat{Y} \rightarrow \hat{Y}'$ is included in $\hat{R}$ if and only if $R \wedge \gamma(\hat{Y}) \wedge \neg\gamma(\hat{Y}')$ is unsatisfiable. Checking unsatisfiability is much easier than enumerating all the satisfying assignments. Moreover, an over-approximation can be easily obtained using this method by restricting the choice of $\hat{Y} \rightarrow \hat{Y}'$ to be considered [66].

Equations (2.5) and (2.6) can be used to compute abstract models for both hardware and software verification. To determine the validity of the proof obligations involved, a general theorem prover, such as Simplify [58], is used. Since variables in hardware designs are usually bit-vectors with small length and the predicates involved in hardware verification are propositional formulas, using a SAT solver, such as zChaff, can be more efficient for hardware verification.

The abstract model built according to equations (2.5) and (2.6) is called the *most accurate abstract model.* Note that, in this abstract model, every abstract initial state has at least one corresponding concrete initial state, and every abstract transition has at least one corresponding concrete transition. However, to build the most accurate abstract model, there are exponential number (in the number of predicates) of implications that need to be checked in worst case. To reduce the abstraction time, in practice an *approximate abstract model* is constructed by intentionally excluding certain implications from consideration. Therefore, there are more behaviors in the approximate model than in the most accurate abstract model. We call the abstract transitions that do not have any corresponding concrete transitions *spurious transitions* (Precise definitions are given in Chapter 5). Since an approximate abstract model contains all the behaviors of the original concrete system, the preservation theorem still holds. In this thesis, to reduce the abstraction time, we restrict $\hat{Y}_1$ and $\hat{Y}'$ to be at most one literal, and restrict $\hat{Y}$ to include at most two literals. The model so obtained will be an over-approximation of the abstract model. We rely on refinement to compute a precise enough abstract model when necessary.

2.3.1 A Software Example

In this subsection, we will illustrate how an abstract program can be generated for a given concrete C program and a set of predicates using the framework presented in [6]. Let $\{P_1, \dots, P_k\}$ denote the given set of concrete predicates. For each predicate P_i, let B_i be the corresponding boolean variable in the abstract program. Let $B = \{B_1, \dots, B_k\}$. A *cube* c in the abstract program is a conjunction $c_1 \wedge \cdots \wedge c_m$, where each literal $c_j \in \{B_j, \neg B_j\}$ for some $B_j \in B$. The concretization of a cube c, denoted $\gamma(c)$, is the conjunction of the concretization of each literal in c. For a statement s and a formula ϕ over the concrete state variables, the *weakest precondition*, denoted $\mathcal{WP}(s, \phi)$, is the weakest predicate whose truth before s entails the truth of ϕ after s terminates. Let $\mathcal{F}_B(\phi)$ denote the largest disjunction of cubes c over B such that $\gamma(c)$ implies ϕ. Given two boolean expressions e and f that are never true simultaneously, we define the function $\mathcal{H}$:

$$\mathcal{H}(e, f) = \begin{cases} \text{true} & \text{if } e \\ \text{false} & \text{if } f \\ \{\text{true}, \text{false}\} & \text{otherwise} \end{cases}$$

To abstract a C program, each line of code is abstracted separately. For each literal over B, the weakest precondition is first calculated, then an expression over B is calculated using $\mathcal{F}_B$ for the weakest precondition. This is illustrated by the following example from [8].

Example 2.3.1 Let $P = \{(x == 1), (x == 2), (x \leq 3)\}$ and let $B = \{B_1, B_2, B_3\}$ be the three corresponding boolean variables. Consider the assignment statement $x := x + 1$. The following table shows the calculation of weakest precondition and the strengthening using the predicates. Based on

	$e = (x == 1)$	$e = (x == 2)$	$e = (x \leq 3)$
$\mathcal{WP}(x{:=}x+1, e)$	$x == 0$	$x == 1$	$x \leq 2$
$\mathcal{F}(\mathcal{WP}(x{:=}x+1, e))$	false	B_1	$B_1 \vee B_2$
$\mathcal{WP}(x{:=}x+1, \neg e)$	$x \neq 0$	$x \neq 1$	$x \geq 3$
$\mathcal{F}(\mathcal{WP}(x{:=}x+1, \neg e))$	$B_1 \vee B_2 \vee \neg B_3$	$\neg B_1 \vee B_2 \vee \neg B_3$	$\neg B_3$

Table 2.1: Predicate abstraction for a C program

this table, the following abstract program is constructed.

$$\begin{aligned} B_1 &:= \mathcal{H}(\text{false}, B_1 \vee B_2 \vee \neg B_3) \\ B_2 &:= \mathcal{H}(B_1, \neg B_1 \vee B_2 \vee \neg B_3) \\ B_3 &:= \mathcal{H}(B_1 \vee B_2, \neg B_3) \end{aligned}$$

2.3.2 A Hardware Example

In this subsection, we present a hardware example and show how it can be verified based on the framework in [66]. Note that, this example is only used to illustrate the traditional predicate abstraction techniques. It does not represent the kind of hardware systems and properties that this thesis is focused on. In fact, we are more interested in verifying the control logic of hardware designs rather than the memory read address calculation described in this example.

This example is a simplified version of the fetch unit of a jpeg encoder implemented using Xilinx FPGAs [45]. This fetch unit is responsible to read each 8 pixels by 8 pixels of grey scale image data and pass it for further processing until the whole image is read (Figure 2.1). The image data is stored in an external memory, where each pixel is represented by one byte. The image data is stored line by line. The fetch unit begins by first reading the width and height of the input grey-scaled image. Noted that the width and height are both multiples of 8. Then starting from memory address 0, 8 pixels by 8 pixels of image data is fetched according to the order following the arrows in Figure 2.1. Essentially the fetch unit has reordered the image data. Given a 8 pixels by 8 pixels image block, the *top-left* pixel is the first

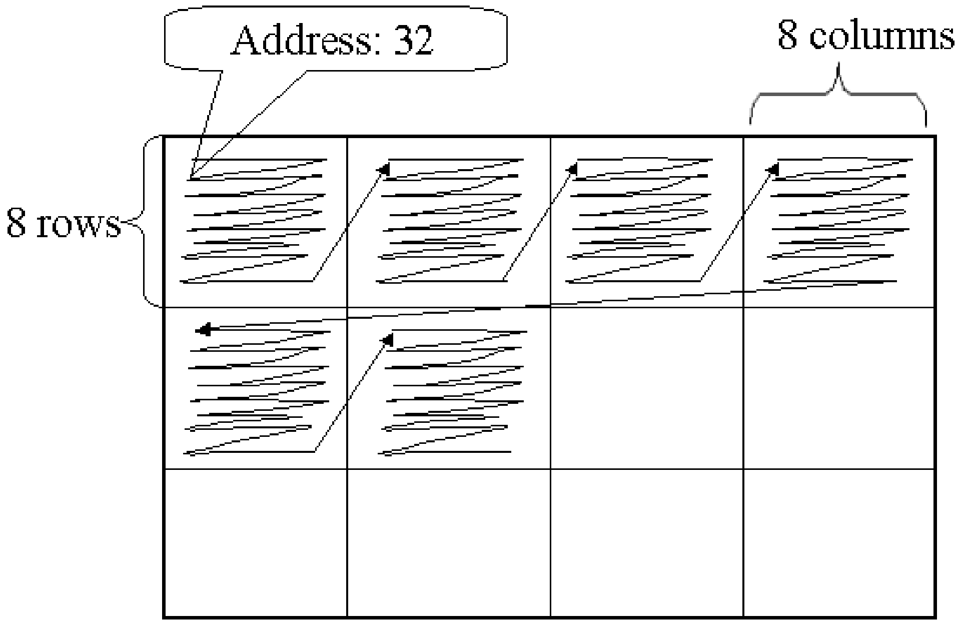

Figure 2.1: Fetch each 8x8 pixel block

pixel of this block. For any pixel in this block, the *left-most* pixel is the first pixel in the same row. Given two adjacent 8x8 image blocks, where the second block is to the right of the first one, the second block is the *next-right block* of the first one. Following is the Verilog implementation. Input signals *WIDTH* and *HEIGHT* are the width and height of the image. We assume they are parameters to the design that do not change. Output signal *addr* is the memory read address. Since the property for this example only concerns the memory read address calculation, the read and write operations of the image data are omitted. Signals *cwidth* and *cheight* are the width and height of the top-left pixel of the current 8x8 block. Signal *row* is the row number within the current 8x8 block. Signal *rowaddr* is the address of the top-left pixel in the next-right block.

```
module fetch(clk, reset, addr, WIDTH, HEIGHT);
input            clk, reset;
output [15:0]    addr;
input [7:0]      WIDTH;
input [7:0]      HEIGHT;

reg [15:0]       addr;
reg [7:0]        cwidth;
reg [7:0]        cheight;
reg [2:0]        row;
reg [15:0]       rowaddr;

always @(posedge clk or posedge reset) begin
        if (reset) begin
            cwidth <= 0;
            cheight <= 0;
            row <= 0;
            addr <= 0;
            rowaddr <= 0;
        end else begin
            if (addr[2:0] < 3'h7)
                addr <= addr + 1;
            else begin
                if (row == 3'h0)
                    rowaddr <= addr + 1;
                if (row < 3'h7) begin
                    row <= row + 1;
                    addr <= (addr & 16'hfff8) + WIDTH;
                end else begin
                    row <= 0;
                    if (cwidth + 8 < WIDTH) begin
                        cwidth <= cwidth + 8;
                        addr <= rowaddr;
                    end else begin
                        addr <= addr + 1;
                        cwidth <= 0;
                        if (cheight + 8 < HEIGHT)
                            cheight <= cheight + 8;
                        else begin
                            cheight <= 0;
                            addr <= 0;
                            row <= 0;
                        end
                    end
                end
            end
        end
end
endmodule
```

The property to be verified is that the read memory address is always smaller than the multiplication of the image width and height. To verify this property using predicate abstraction, the following signals and predicates are identified manually:

- Signal row[2:0]. It represents the row number of the current pixel within its 8x8 block.
- Signal addr[2:0]. It is the lowest 3 bits of the signal *addr*. It represents the column number of the current pixel within its 8x8 block.
- Predicate “addr & 16’hfff8 = (cheight+row) * WIDTH + cwidth”. It is an invariant, which says that the address of the left-most pixel is the addition of (row * WIDTH) and the address of the top-left pixel which is (cheight * WIDTH + cwidth).

- Predicate "rowaddr = cheight * WIDTH + cwidth + 8". It is not an invariant, which is only true when (row >= 1). The predicate says that, the address of the top-left pixel in the next-right 8x8 block (If one exists), is 8 more than the address of the top-left pixel of the current 8x8 block. Because signal *rowaddr* does not get the correct value until the end of the first row, so until then, this predicate is false in the design.
- Predicates "cwidth + 8 < WIDTH" and "cheight + 8 < HEIGHT". These two predicates are branch conditions in the program.
- Predicate "addr < WIDTH * HEIGHT". It is the property to be proven.
- Besides the above predicates, we assume the following invariants in building the abstract model. The correctness of this assumption can be checked easily by a syntactic analysis of the given Verilog code.
  - 8 divides cwidth
  - 8 divides cheight
  - cwidth < WIDTH
  - cheight < HEIGHT

Note that, signals row[2:0] and addr[2:0] are retained in the abstraction, so that their initial states and transition relations are copied from the concrete model. The algorithms presented in [66] works for guarded command languages. It is easy to translate the above Verilog code into a guarded command language based program. The corresponding abstract model expressed in SMV language is shown below:

```
MODULE main
VAR
        row : 0..7;
        addr2_0 : 0..7;
        p_addr : boolean; –addr & 16'hfff8 = (cheight+row) * WIDTH + cwidth
        p_rowaddr : boolean; –rowaddr = cheight * WIDTH + cwidth + 8
        p_cwidth2 : boolean; –cwidth + 8 < WIDTH
        p_cheight2 : boolean; –cheight + 8 < HEIGHT
        prop : boolean; –addr < WIDTH * HEIGHT

ASSIGN
init(row)         := 0;
init(addr2_0)     := 0;
init(p_addr)      := 1;
init(p_rowaddr)   := 0;
init(p_cwidth2)   := {0,1};
init(p_cheight2)  := {0,1};
init(prop)        := 1;

next(p_rowaddr) :=
        case
        addr2_0<7 : p_rowaddr;
        row=0 : case p_addr: 1; 1: 0;  esac;
        row < 7 : p_rowaddr;
        1: {0,1};
        esac;

next(addr2_0)    :=
        case
        addr2_0 < 7: addr2_0+1;
        row < 7: 0;
        p_cwidth2:
              case p_rowaddr: 1; 1: {0,1,2,3,4,5,6,7}; esac;
        1: 0;
        esac;

next(p_addr)     :=
        case
        addr2_0 < 7: case p_addr: 1; 1: {0,1}; esac;
        row < 7: case p_addr: 1; 1: {0,1}; esac;
        p_cwidth2: case p_rowaddr: 1; 1: {0,1}; esac;
        p_cheight2: case p_addr: 1; 1:{0,1}; esac;
        1: 1;
        esac;

next(row)        ::=
        case
        addr2_0<7: row;
        row<7: row+1;
        1: 0;
        esac;

next(p_cwidth2) ::=
        case
        addr2_0<7 | row<7: p_cwidth2;
        p_cwidth2: {0,1};
        1: 1;
        esac;

next(p_cheight2)::=
        case
        addr2_0<7 | row<7 | p_cwidth2: p_cheight2;
        p_cheight2: {0,1};
        1: 1;
        esac;

next(prop)      ::=
        case
        addr2_0<7: case prop: 1; 1: {0,1}; esac;
        row < 7: case p_addr: 1; 1: {0,1}; esac; –p_addr, cwidth<WIDTH, cheight<HEIGHT,
                                                  – 8 divides HEIGHT, 8 divides cheight
        p_cwidth2: case p_rowaddr: 1; 1: {0,1}; esac;
        p_cheight2: case p_addr: 1; 1: {0,1}; esac;
        1: 1;
        esac;


SPEC AG (prop)
```

We have verified that this abstract model satisfies the required property, thus the concrete Verilog program satisfies the same property.

2.4 Localization Reduction

Localization reduction [43] is also a special case of existential abstraction. In localization reduction, a set of important state variables, called *visible* variables, are retained in the abstract model; while the rest, called *invisible* variables, are left unconstrained (Their values are assigned nondeterministically). The abstract transition is obtained by conjuncting the transition relations for the visible variables. Formally, let V be the set of concrete state variables, and S be the concrete state space. For each state $s \in S$, the value of a variable $v \in V$ in state $s \in S$ is denoted by $s(v)$. In localization reduction, given a set of visible variables $V_1 \subseteq V$, the abstract state variables $U = \{u_1, u_2, \ldots, u_k\}$ satisfies $U \subseteq V \wedge U \supseteq V_1$. The set of abstract states for localization reduction is $\hat{S} = D_{u_1} \times D_{u_2} \ldots \times D_{u_k}$. The simulation relation is $\rho(s, \hat{s}) = (\mathit{proj}[U](s) \equiv \hat{s})$. Given an abstract state $\hat{s}$, the set of related concrete states is $\gamma(\hat{s}) = \{s | \mathit{proj}[U](s) \equiv \hat{s}\}$.

We also assume that neither the concrete transition relation nor the set of initial states is described as a single formula. Instead, for each individual variable $v \in V$, the transition relation of v is represented as a propositional formula R_v and the set of initial states of v is represented as a propositional formula I_v. The *most accurate abstract model* for localization reduction can be easily built. That is, the abstract initial states $\hat{S_0}$ and the abstract transition relation $\hat{R}$ are defined as

$$\hat{S_0} = \wedge_{v \in U} I_v \tag{2.7}$$

$$\hat{R} = \wedge_{v \in U} R_v \tag{2.8}$$

It is usually the case that $\hat{R}$ depends not only on current and next state variables on U, but also some invisible variables which occur in some R_v or I_v. In the abstract model, these invisible variables are treated as primary inputs. In general, the size of the abstract transition relation may be large since it is directly copied from the concrete model. In Section 4, we will show techniques to reduce the size of the abstract model using approximation.

2.5 Abstraction Refinement

Existential abstraction is a conservative approach for model checking universal temporal logic [24] properties (we only consider safety properties in this thesis). That is, the correctness of any universal formula on an abstract system automatically implies the correctness of the formula on the concrete system. However, a counterexample on an abstract system may not correspond to any real path, in which case it is called a *spurious* counterexample [23]. To get rid of a spurious counterexample, the abstraction needs to be made more precise via refinement. *Counterexample guided abstraction refinement* [23, 38, 72] (CEGAR) automates this procedure. It works as follows: For a given system, an abstract model that is guaranteed to include all behaviors of the original system is built. Model checking is then applied to the abstract model. If the property holds, it is true of the concrete model and verification terminates. In case the property is violated on the abstract model a counterexample is generated. This abstract counterexample is checked against the concrete model. If the abstract counterexample corresponds to a concrete execution path, the property is proved to be false and verification terminates. Otherwise, the abstract counterexample is *spurious* and it is used to guide the refinement of the abstract model. The above procedure repeats until the property is confirmed or refuted. Figure 2.2 shows the fours steps in the above abstraction refinement framework.

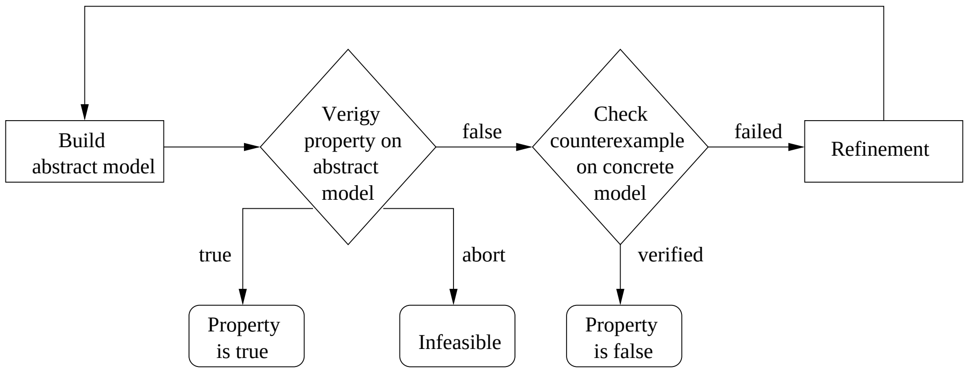

Figure 2.2: General Abstraction Refinement framework

Chapter 3

SAT and Unsatisfiability Proofs

In this chapter, we first briefly review Davis-Putnam-Logeman-Loveland (DPLL) backtracking SAT algorithms with conflict learning. Then, an unsatisfiability proof extraction algorithm is presented. Based this algorithm, we explain how to implement an incremental SAT solver.

3.1 Conflict based Learning in SAT Solvers

In this section, we briefly describe the learning mechanism used by modern SAT solvers, such as GRASP [68], Chaff [56,76] and BerkMin [31]. These SAT solvers are based on the Davis-Putnam-Logeman-Loveland (DPLL) backtracking SAT algorithm.

Let BV be a finite set of boolean variables. For any variable $v \in BV$, recall that a literal l_v over v is either v or $\neg v$. A *clause* c is a finite disjunction of literals. A *CNF formula* f is a finite conjunction of clauses. For convenience, we represent a CNF formula f by the set of clauses in f. A set of clauses $\mathbf{E}_f$ which includes all the clauses of f and some other clauses that are logically implied by f is called an *extension* of f. Formally

$$(f \subseteq \mathbf{E}_f) \wedge (\mathbf{E}_f \subseteq \{c \mid \text{c is a clause and } f \Rightarrow c\}).$$

It is easy to show that f is equivalent to any of its extensions.

An *assignment* $\mathcal{A}$ is a partial function from BV to $\{0,1\}$. If $v \in BV$ is not in the domain of $\mathcal{A}$, the value of v under $\mathcal{A}$ is undetermined. An assignment $\mathcal{A}$ is *complete* if it assigns a value to every variable.

Boolean constraint propagation (BCP for short) is an essential component of DPLL based SAT solvers. Given a clause c and an assignment $\mathcal{A}$, if the value of c under $\mathcal{A}$ is undetermined and the value of only one literal l_v in c is undetermined under $\mathcal{A}$, then c is called a *unit clause* and l_v is called a *unit literal*. The rest of the literals in c are called the *antecedents* of l_v, denoted collectively by $\mathbf{ant}(l_v, c)$. For a unit clause that has only one literal, the antecedent set of the literal is empty. Given an assignment $\mathcal{A}$, where clause c is a unit clause and l_v is the unit literal in c, BCP extends $\mathcal{A}$ by mapping l_v to 1. The resulting assignment is called an *extension* of $\mathcal{A}$. The correctness of the BCP algorithm relies on the fact that given a unit clause c under $\mathcal{A}$, only those extensions of A with $l_v = 1$ can make c be 1. Such an assignment of a value to a literal is called an *implied assignment*. Not all assignments need to be implied assignments, some of them might be "guessed" by the SAT solver. Such assignments are called *decision assignments*. For a partial assignment $\mathcal{A}$ we define $\mathcal{A}_d$ to be the set of decision assignments in $\mathcal{A}$.

Given an extension g of a CNF formula f, a partial assignment $\mathcal{A}$ and the set of decision assignments $\mathcal{A}_d \subseteq \mathcal{A}$, an *implication graph* $IG(g, \mathcal{A}_d) = \langle \mathcal{X}, \mathcal{E} \rangle$ is a directed acyclic graph, where

- Each vertex $x \in \mathcal{X}$ is labeled with a literal. The label is denoted by $\mathcal{L}(x)$.

- For each literal l in $\mathbf{ant}(\mathcal{L}(x), c)$, where c is a unit clause and $\mathcal{L}(x)$ is the unit literal in c, there will be a predecessor vertex of x labeled by that literal l. For each predecessor p of x there is a directed edge $e \in \mathcal{E}$ that starts from p and ends at x and is labeled with the unit clause c.

- A vertex x is a root of the graph, if there are no incoming edges for x. So each decision assignment in $\mathcal{A}$ is the label of some root vertex. The label of each vertex x, that is not a root, is implied by other assignments in $\mathcal{A}$.

- We associate with each decision assignment a number greater than or equal to 1, called the *decision level*. When we add a new decision assignment l_v to a set of decisions $\mathcal{A}$, the decision level of l_v is one more than the maximum decision level in $\mathcal{A}$. The decision levels for implied assignments will be determined by decision levels of other assignments in $\mathcal{A}$. For a non-root vertex x, let c be the unit clause that implies the value of label $\mathcal{L}(x)$ of x. The decision level of $\mathcal{L}(x)$ is the maximum of the decision levels of its antecedent literals.

- When some clause c evaluates to 0 under $\mathcal{A}$, we introduce a new vertex κ, and label it with *false*. We also add an edge from each vertex labeled with a literal in c to κ, and label each edge with c. c is called the *conflicting clause.*

When an implication graph $IG(g, \mathcal{A}_d)$ includes the vertex κ, we call the subgraph that can reach κ a *conflict graph.* A CNF formula f is unsatisfiable, if there exists an extension g of f and a conflict graph, $IG(g, \emptyset)$, where the set of decision assignments is empty. Intuitively, any assignment implied in this graph, including the label *false* of the vertex κ, is derived from the current CNF formula g without any decisions. This means g is unsatisfiable. Thus f itself is unsatisfiable because g is unsatisfiable if and only if f is unsatisfiable.

Given a conflict graph $IG(\mathbf{E}_f, \mathcal{A}_d)$, let $\Omega(IG)$ be the set of clauses that label edges in IG. If $\mathcal{A}_d \neq \emptyset$, there exists at least one vertex cut $CUT = \{x_1, \ldots, x_n\}$, that separates decision variables and the conflict vertex κ. The vertices of the cut can have both decision and implied assignments as labels. Denote the subgraph obtained from IG by dropping vertices on the decision variable side of the cut by IG_{CUT}. Let the clause corresponding to CUT be

$$cl(CUT) = \bigvee_{1 \leq i \leq n} \neg \mathcal{L}(x_i).$$

This clause is called a *conflict clause.*

Lemma 3.1.1 *Let IG be a conflict graph, CUT be a vertex cut of IG that separates the conflict vertex with the decision vertices. Then*

$$(\bigwedge \Omega(IG_{CUT})) \Rightarrow cl(CUT).$$

Proof: It suffices to prove that $(\bigwedge \Omega(IG_{CUT})) \wedge \neg cl(CUT) \Rightarrow$ false. We prove this by induction over the size of IG_{CUT}.

- For the base case, IG_{CUT} only includes the conflicting clause c that directly leads to the conflict vertex κ. It is easy to see that $\Omega(IG_{CUT}) = \{c\}$ and $cl(CUT) = c$. Thus the conjunction is false.

- Induction step. We need to prove the lemma for a vertex cut, CUT. For BCP to work, there is at least one clause c_1, where all but one literal v_1 of it whose negations are in this cut. Thus there exist a cut $CUT_1 = CUT - \{l | \neg l \in c_1\} \cup \{v_1\}$ in IG. Since $\Omega(IG_{CUT_1}) = \Omega(IG_{CUT}) \setminus \{c_1\}$, according to the induction hypothesis $(\bigwedge \Omega(IG_{CUT_1})) \wedge \neg cl(CUT_1) \Rightarrow$ false holds. It is easy to see that clause c_1 together with the literals in CUT imply v_1. Thus the lemma holds for the CUT.

∎

For example, the proof in the induction step can be illustrated using the conflict graph A in Figure 3.2. The cut x_2, x_{15}, x_9 is implied by the cut $\neg x_{11}, x_{15}, x_9$ and clause $\omega_1 = x_{11} \vee \neg x_{15} \vee x_2$.

Based on Lemma 3.1.1, if $\Omega(IG_{CUT})$ is a subset of an extension of f, then $f \Rightarrow cl(CUT)$. *Conflict-based learning* starts with f, and gradually extends f by adding conflict clauses identified during the search. It is easy to prove by induction that the set of clauses at any time during the SAT search is always equivalent to f. Note that there will be no decision assignments in the final conflict graph of an unsatisfiable formula. For ease of presentation, we associate an empty cut with the last conflict graph, where the set of decisions $\mathcal{A}_d = \emptyset$. The corresponding conflict clause is the empty clause, which is logically equivalent to *false*.

It is possible to generate more than one conflict clause from a single conflict graph [68]. Each clause corresponds to a different cut of the conflict graph. Among the set of conflict clauses generated from a single conflict graph, there must be at least one conflict clause $cl(CUT)$ that contains only one literal l_v from the maximum decision level [76]. Such a conflict clause is called *asserting clause* [76]. The vertex corresponding to l_v in the conflict graph is called *unique implication point* (UIP). Note that the cut consisting of all decision assignments corresponds to one such conflict clause. The vertex cut including a UIP vertex that is the closest to the conflict vertex κ is called the *first UIP cut* (1UIP cut). For an asserting clause cl, the maximum decision level of the literals in cl, other than l_v, is the *backtrack level* and will be denoted by *blevel*. The SAT solver will backtrack to the decision at decision level *blevel* and remove all those assignments whose decision levels are greater than *blevel*. This will make cl a unit clause and l_v a unit literal. Thus, l_v which was previously a decision assignment now becomes an implied assignment. This process is referred to as *non-chronological backtracking.*

```
while (choose_decision()) {      // Decision
  while (BCP() == conflict) {    // Propagate implications
    blevel = analyse_conflict(); // Conflict learning
    if (blevel == 0)             // No decisions
      return UNSAT;
    else
      backtrack(blevel);    // Non-chronological backtrack
  }
}
return SAT;                 // All vars have been assigned
```

Figure 3.1: Basic DPLL backtracking search

3.2 SAT Conflict Dependency Analysis

In this section we present a technique called *conflict dependency analysis*. When a SAT solver concludes that a given CNF formula is unsatisfiable, our technique can efficiently identify a subset of the conflict graphs generated during the SAT search as the proof of unsatisfiability. The set of clauses in the extracted proof is itself unsatisfiable. Often the new set of clauses is significantly smaller than the original set of clauses. An incremental SAT solver is developed based on conflict dependency analysis.

3.2.1 Dependencies between Conflict Graphs and Clauses

Definition 3.2.1 Given two conflict graphs A and B, if at least one of the conflict clauses generated from A labels one of the edges in B, then we say that conflict graph B *directly depends* on conflict graph A.

For example, consider the conflicts depicted in the conflict graphs of Figure 3.2. Suppose that at a certain stage of the SAT checking, conflict graph A is generated. This produces the conflict clause $\omega_9 = (\neg x_9 + x_{11} + \neg x_{15})$. We are using the first UIP (1UIP) learning strategy [76] to identify the conflict clause here. This conflict clause can be rewritten as $x_9 \wedge \neg x_{11} \rightarrow \neg x_{15}$. In the other conflict graph B, clause ω_9 labels one of the edges, and forces variable x_{15} to be 0. Hence, we say that conflict graph B directly depends on conflict graph A.

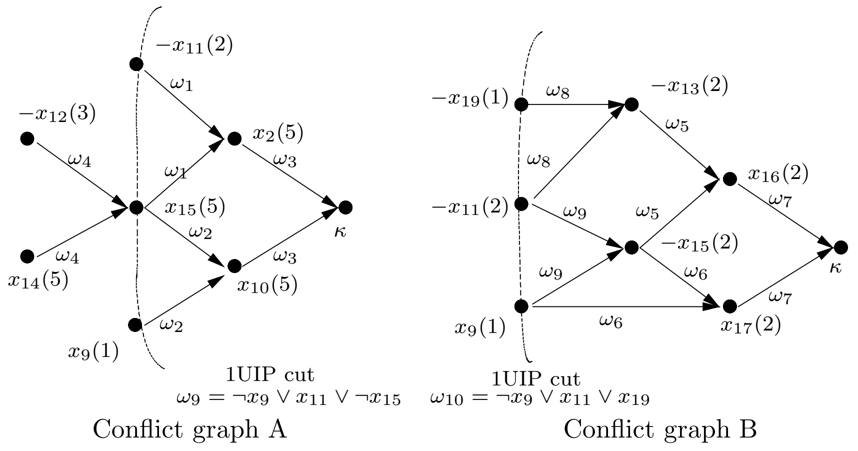

Figure 3.2: Two dependent conflict graphs

Given the set of conflict graphs generated during satisfiability checking, we construct the *conflict dependency graph* as follows:

- Vertices of the dependency graph are all conflict graphs created by the SAT algorithm.
- Edges of the dependency graph are direct dependencies.

Figure 3.3 shows an conflict dependency graph with five conflict graphs. A conflict graph B depends on another conflict graph A, if vertex A is reachable from vertex B in the dependency graph. In Figure 3.3, conflict graph E depends on conflict graph A. When the SAT algorithm detects unsatisfiability, it terminates with the last conflict graph corresponding to the last conflict.

Definition 3.2.2 The *proof graph* is a subgraph of the conflict dependency graph. It includes the last conflict graph and all the conflict graphs on which the last one depends.

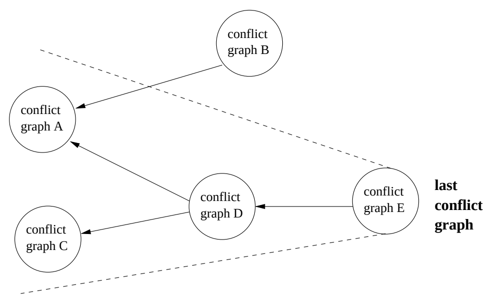

Figure 3.3: The conflict dependency graph and the proof graph (within dotted lines)

In Figure 3.3, conflict graph E is the last conflict graph, hence the proof graph includes conflict graphs A, C, D, E. For an unsatisfiable CNF formula, the proof graph can be constructed from the conflict dependency graph by any directed graph traversal algorithm for reachability. Typically, many conflict graphs can be pruned away in this traversal, so that the proof graph becomes much smaller than the dependency graph. Intuitively, all SAT decision strategies are based on heuristics. For a given SAT problem, the initial set of decisions/conflicts a SAT solver comes up with may not be related to the final unsatisfiability result. Our dependency analysis helps to remove that irrelevant reasoning. For an unsatisfiable CNF formula, we call the proof graph the unsatisfiability proof. It is easy to verify the validity of a given proof graph. Any conflict clause in the proof graph must be associated with a conflict graph. Each conflict graph must logically leads to false as required by Lemma 3.1.1.

Using the proof graph, we can identify the part of a given unsatisfiable CNF formula that the SAT solver uses to prove unsatisfiability. Intuitively, the set of clauses in the proof graph is enough to determine unsatisfiability. Recall from Section 3.1 that each conflict clause $cl(CUT)$ corresponds to a vertex cut CUT. We associate an empty cut and an empty clause (denoted as θ) with the last conflict graph which does not have any decision assignments. As mentioned in Section 3.1, $(\bigwedge \Omega(IG_{CUT})) \Rightarrow cl(CUT)$. In the following f stands for the CNF formula under consideration.

A conflict clause $cl(CUT)$ *directly depends* on a clause b iff b is one of the clauses in $\Omega(IG_{CUT})$. We say the conflict clause a *depends* on clause b iff there exist $a = c_1, c_2, \ldots, b = c_n$, such that for $1 \leq i < n$, c_i directly depends on c_{i+1}. The set of clauses in f that a given set of conflict clauses *cls* depend on is called the *dependent set* and the set is denoted by $dep(cls)$.

For example, consider the conflict graphs in Figure 3.2 (only parts of the conflict graphs that are relevant are shown). The conflict clause $c_9 = (\neg x_9 \vee x_{11} \vee \neg x_{15})$, which corresponds to the 1UIP (see [76]) cut of A, *directly depends* on the clauses c_1, c_2 and c_3. The conflict clause $c_{10} = \neg x_9 \vee x_{11} \vee x_{19}$ is generated based conflict graph B and clause c_9 labels one of the edges in the subgraph of the 1UIP cut in B. Hence, we say that conflict clause c_{10} *directly depends* on conflict clause c_9. The other clauses on which c_{10} *directly depends* are c_5, c_6, c_7 and c_8. Note that the clause c_9 need not be a clause in the original CNF formula. Since c_9 *directly depends* on c_1, c_2 and c_3, it follows c_{10} *depends* on c_1, c_2 and c_3.

For an unsatisfiable CNF formula f, the SAT solver will end with an extension g of f which is logically equivalent to $false$ and an empty conflict clause θ. If a conflict clause is deleted during the SAT search, we add it to g. For each conflict clause $cl \in (g \setminus f)$, we maintain the set of clauses that it directly depends on. Note that g and the dependencies among the clauses are determined by the conflict graphs and conflict clauses generated by the

SAT solver.

Denote by $SUB(f)$ those clauses of f that appear in the dependent set of the empty conflict clause θ. If we let $\{f\}$ denote the set of clauses in f then $SUB(f) = dep(\theta) \cap \{f\}$. Note that $SUB(f)$ will have clauses from f alone, none of the conflict clauses are in $SUB(f)$. The following theorem states that $SUB(f) \subseteq f$ is itself unsatisfiable.

Theorem 3.2.1 *Let* $cls \subseteq g$ *be a set of conflict clauses, then* $(\bigwedge(dep(cls) \cap \{f\})) \Rightarrow (\bigwedge cls)$. *In particular,* $SUB(f)$ *is unsatisfiable.*

In general, for an unsatisfiable CNF formula f, $SUB(f)$ may not be the minimal unsatisfiable subset of f, but it can be substantially smaller than f. This is because the set of conflict graphs in the proof graph is only a subset of all the conflict graphs, and for each conflict graph, only a subset of the clauses in it is included in $SUB(f)$.

Maintain Dependencies Based on Zchaff

We have implemented the conflict dependency analysis algorithm on top of zchaff [76], which has a powerful learning strategy called first UIP (1UIP). Experimental results from [76] show that 1UIP is the best known learning strategy. In 1UIP, only one conflict clause is generated from each conflict graph, and it only includes those implications that are closer to the conflict. Refer to [76] for the details. We have built our algorithms on top of the 1UIP learning strategy. In the following, we only deal with the case that only one conflict clause is generated from a conflict graph. Note here that our algorithm can be easily adapted to other learning strategies.

We assign a unique identifier, *cid*, for each clause appearing in the SAT search, which could be either an original clause or a conflict clause. In our case, the *cid* of a conflict clause is also used to represent the corresponding conflict graph. During the SAT process, once a conflict is analyzed and conflict clauses are generated, the corresponding conflict graph is deleted. To be able to analyze the conflict graphs when the SAT solver terminates, we store the information of all conflict graphs in a file. For the last conflict graph, we store the identifiers of the clauses appearing in the whole conflict graph; while for the other conflict graphs, only those clauses between the 1UIP cut to the conflict vertex are stored. For example, for conflict graph A in Figure 3.2, only clauses $\omega_1, \omega_2, \omega_3$ are stored. The reason to leave out clause ω_4 is because, ω_1, ω_2 and ω_3 are enough to prove that the assignments $-x_{11}, x_9, x_{15}$ will definitely lead to a conflict. Since the 1UIP cut is very close to the conflict vertex in a conflict graph, this reduces the size of the generated proof graph and the size of the unsatisfiable subformula.

After SAT terminates with unsatisfiability, our pruning algorithm starts from the last conflict graph. Based on the clauses contained in this conflict graph, the algorithm traverses other conflict graphs that this one depends on. The result of this traversal is the proof graph and the unsatisfiable subformula. Note that, the dependencies we extracted are not affected by whether the CNF formula is satisfiable or not. The knowledge about what are the clauses that one conflict clause depends on help us to decide whether to keep this conflict clause, when solving a new SAT instance. This is explained in the next subsection.

3.2.2 Incremental SAT

A SAT solver with conflict-based learning derives conflict clauses to avoid repeating the same contradictory assignments. Although the resulting CNF formula is logically equivalent to the original formula f, the added conflict clauses record the searches that the SAT solver has already tried. In fact, modern solvers rely on periodic random restarts to avoid getting stuck due to bad decisions made early in the search. In this case, the current set of decisions is thrown away, and the SAT solver relies on the learned conflict clauses to save the previous search efforts.

If two SAT problems f_1 and f_2 have a significant number of clauses in common then the conflict clauses learned while solving one of them may be useful in solving the other. In particular, if the dependent set of a conflict clause c learned while solving the first SAT problem is a subset of f_2, i.e., $dep(\{c\}) \cap \{f_1\} \subseteq \{f_2\}$ then the conflict clause c will be useful while solving the second SAT problem. Although, there is some overhead in maintaining the dependencies between conflict clauses while solving f_1, in practice we have found that this speeds up SAT solver while handling f_2.

Chapter 4

Localization Reduction

In this chapter, we present techniques for localization reduction that can verify real-world designs containing thousands of registers.

As presented in Section 2.4, the most accurate abstract models for localization reduction can be easily built. However, the size of the abstract models can be large, if the next state logic of some visible variables are large (For a gate level circuit, the next state logic of a visible variable is the gates in the direct fan-in of this variable). To keep an abstract model small, we construct an overapproximation to the most accurate abstract model.

The refinement algorithm for localization reduction computes a small set of invisible variables and makes them visible. As a result, the current spurious abstract counterexample is invalidated. For hardware designs with thousands of registers, to compute such a set of invisible variables is challenging. We present two refinement algorithms. The first one is based on the fact that, if the given spurious counterexample requires an invisible variable to be assigned a specific value, then this invisible variable is important, and should become a candidate for refinement. The second algorithm makes an invisible variable visible if it appears in the unsatisfiability proof generated during checking the spurious counterexample on the concrete model.

4.1 Overapproximate the Abstract Models

In this section, we review the overapproximation method presented by Ho et.al. in [40]. For a gate level circuit, we call the most accurate abstract model built using equations (2.7) and (2.8) the *no-cut model*. Abstraction in localization reduction starts with a subset of important registers. In the initial abstraction, this subset consists of signals appearing in the property to be proven. Later, each refinement step adds more registers to this subset. We also refer to these important registers as visible registers, included registers, or selected latches. Given a set of important registers, the primary inputs and the output signals of registers that are the inputs to the next state logic of these important registers are called the *no-cut signals*. A no-cut model includes the important registers, the no-cut signals, and the combinational logic between them. The signals in no-cut which are the output signals of included registers are called *bound inputs*, the other signals in no-cut are called *free inputs*. The signals in free inputs which are the output signals of invisible registers in the concrete model are called *excluded registers* (or non-selected latches). Not that, only those invisible registers that directly feed into visible registers are the excluded registers. In our experience, given 50 important registers, the number of free inputs in the corresponding no-cut model can be over a thousand. Each of these free inputs becomes an abstract variable during abstract model checking. This may not affect the performance of forward image computation very much, because the free inputs are quantified using the early quantification algorithm [15, 69]. However, many inputs makes the counterexample generating much harder during the backward image computation, because the BDDs during the abstract counterexample generation phase have these inputs as the BDD support which can not be quantified.

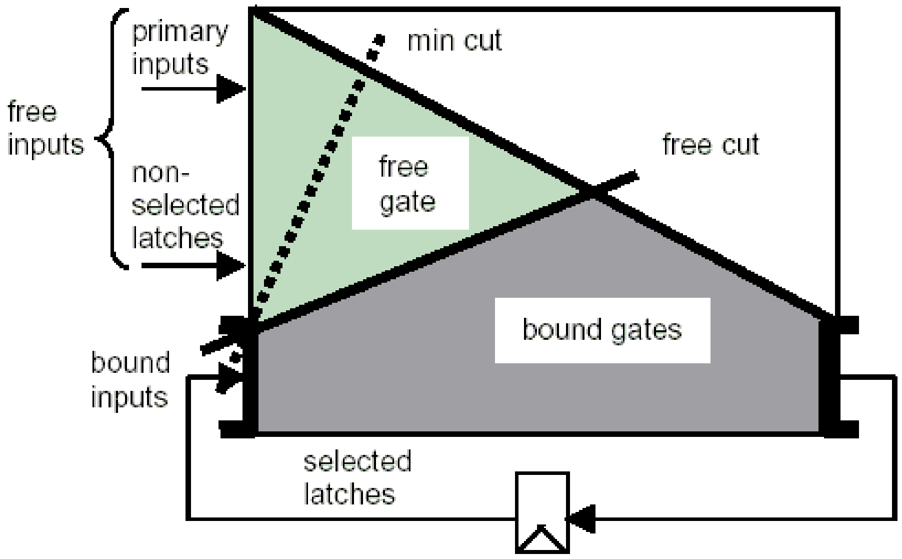

Figure 4.1: no-cut and min-cut abstract models (from [40])

To reduce the size of a no-cut abstract model while adding fewer extraneous behaviors, a *min-cut* abstract model is built. Intuitively, part of the combinational logic that only depends on the free inputs are removed, so that the number of inputs and the number of gates in the min-cut model can be much less than those in the corresponding no-cut model. A gate in the no-cut model is called a *bound gate* if at least one of the bound inputs are in the transitive fan-in of this gate; otherwise it is a *free gate*. The signals that separate the free gates and the bound gates are the *free cuts*. The output of a free gate is in the free cuts, if it is one of the inputs for a bound gate. We will ignore the correlations between the signals in free cuts by removing the combinational logic between free inputs and free cuts. This will introduce extraneous behaviors. But since the removed gates do not depend on any bound inputs, these introduced behaviors usually won't affect the verification of the given property. It is usually the case that the complexity for the model checking of a gate level circuit is more sensitive to the number of state variables rather than the number of gates in it. Therefore, we use a graph min-cut algorithm to find a minimal set of signals that separate the free inputs and the free cuts, and only remove the combinational logic between the free inputs and this min cut. The resulting circuit becomes the min-cut abstract model. The inputs of the min-cut model can be classified into 4-tuple $\langle$*pis*, *excluded*, *cuts*, *included*$\rangle$ (See Figure 4.1), where *pis* are the indeed primary inputs, *excluded* are the *excluded registers*, *included* are the bound inputs and *cuts* are the internal circuit cut points which are generated using the min-cut algorithm. In practice, the number of no-cut signals can be 5 times bigger than the number of min-cut signals. However, the number of min-cut signals can still be around several hundred. In the min-cut model, the ratio of model inputs to model registers can vary between different designs, for example from 0.46 to 36. It is generally the case that having a large number of inputs slows down model checking the circuit. In Section 4.3.2, we present an algorithm to deal with this problem.

4.2 Checking the Validity of an Abstract Counterexamples

Given an abstract model $\hat{M}$ and a safety formula ϕ, we run the usual BDD based symbolic model checking algorithm to determine if $\hat{M} \models \phi$. Suppose that the model checker produces an abstract path counterexample $ce = \langle ce_0, ce_1, \ldots, ce_n \rangle$. To check whether this counterexample holds on the concrete model M or not, we symbolically simulate M beginning with the initial state using a fast SAT checker. At each stage of the symbolic simulation, we constrain the values of variables only according to the given counterexample. The equation for symbolic simulation is:

$$\begin{aligned}(S_0(s_0) \wedge \gamma(ce_0)(s_0)) \quad \wedge \quad & (R(s_0, s_1) \wedge \gamma(ce_1)(s_1)) \wedge \ldots \\ & \wedge (R(s_{n-1}, s_n) \wedge \gamma(ce_n)(s_n)) \end{aligned} \tag{4.1}$$

If this propositional formula is satisfiable, then we can successfully simulate the counterexample on the concrete machine, thus $M \not\models \phi$. The satisfiable assignments to all the variables give a valid counterexample on the concrete model. If this formula is not satisfiable, the counterexample is *spurious* and the abstraction needs to be refined.

4.3 Invisible Variables In Abstract Counterexamples

In this section, an abstraction refinement framework, called *RFN* [72], for the verification of safety properties on gate-level circuits using localization reduction is presented. We design RFN to verify gate-level circuits synthesized from real-world RTL designs containing approximately 5,000 registers, which represents an order of magnitude capacity improvement over previous results. The main feature of RFN is that it uses the spurious abstract counterexamples to identify the candidates for refinement. RFN also combines three different engines (BDD, ATPG and simulation) to handle large circuits using abstraction and refinement.

4.3.1 Guided SAT/ATPG

To check whether an abstract counterexample corresponds to a real concrete path or not, symbolic simulation is performed on the concrete model. During the symbolic simulation, the search space is restricted within the given abstract counterexample, thus this procedure is very fast. However, a real counterexample can only be found if it exactly corresponds to the given abstract counterexample. We introduce *guided SAT/ATPG* to balance the computation time with the chances to find real counterexamples.

In localization reduction, the free inputs, which correspond to the excluded registers and the internal cut points, are unconstrained. Therefore, their values in an abstract counterexample are also arbitrary, which can trivially invalidate the abstract counterexample on the concrete model. For a spurious abstract counterexample, there may exist a slightly different abstract counterexample, where the free inputs are assigned differently, which corresponds to a real counterexample. To search real counterexamples that do not exactly correspond to a given abstract counterexample efficiently, a new algorithm called *Guided-SAT/ATPG* is developed. Instead of applying the entire abstract counterexample to the symbolic simulation, this algorithm gradually incorporates more constraints from the abstract counterexample until a definite answer is obtained in the given time limit. That is, depending on how long the symbolic simulation is allowed, the search space is automatically adjusted by some restrictions to the given abstract counterexample. Therefore, using this algorithm it is more likely to find a real counterexample. At the same time, enough restrictions are obtained from the given abstract counterexample to achieve a reasonable speed. Figure 4.2 shows the detailed algorithm. In Figure 4.2, *model* is the concrete model,

```
Algorithm Guided-SAT (model, counterexample, goal)
    backtrack = 500
    constr_size = 3
    do
        constraints = extract (counterexample, constr_size)
        search = SAT (model, constraints, goal, backtrack )
        backtrack *= 1.5
        constr_size *= 1.5
    while search = abort
    return search
```

Figure 4.2: Guided-SAT Algorithm

counterexample is an abstract counterexample, *goal* is the set of reached states that violate the given property, *backtrack* is the maximal number of backtracks SAT is allowed to try, after which it will return with *abort*. Note that each implication conflict corresponds to a backtrack, thus this number controls the run time of a SAT solver. Variable *constr_size* is the number of assignments from each time frame of the abstract counterexample that will be used as guidance in the SAT search. Function **extract** returns the *constr_size* number of assignments at each time frame. To reduce the noise in free inputs, we apply assignments from excluded registers and cut points only after the assignments from the real primary inputs and included registers have been applied to the SAT search. Function **SAT** uses the obtained partial constraints in satisfiability. Note that *Guided-ATPG* can be implemented by replace **SAT** with **ATPG** [3] in Figure 4.2. In our experience, Guided SAT/ATPG is usually able to run very fast for searching counterexamples of length about one hundred, even with a small amount of guidance, for example when *constr_size* equals 5.

4.3.2 Efficient Abstract Model Checking

BDD based model checkers are used to verify the given property on the abstract model. If the property holds, it also holds on the concrete model. Otherwise an abstract counterexample is generated. As is presented in Section 4.1, since a min-cut model can be much smaller than the corresponding no-cut model, we use min-cut abstract models for abstract model checking. Consequently, the abstract counterexample can have assignments to internal cut point signals generated using the min-cut algorithm. Since RFN examines the excluded registers which are assigned in the abstract counterexamples for refinement, having cut point signals in the counterexample hinders the refinement process. We avoid this problem by mapping (or lifting) the abstract counterexample generated when model checking a min-cut model to an abstract counterexample over the corresponding no-cut model. Given a min-cut abstract counterexample *min_counterexample*, which leads to an *error_state* on the min-cut abstract model, the lifting to no-cut abstract model is achieved by calling **Guided-ATPG**(*no-cut model*, *min_counterexample*, *error_state*). It is usually the case that, Guided ATPG can find a corresponding abstract counterexample on the no-cut model. This is because of two reasons: First, the sets of behaviors between a min-cut model and the corresponding no-cut abstract model are similar. Furthermore, Guided ATPG only uses partial constraints from the min-cut abstract counterexample during the search. However, it is possible that no corresponding abstract counterexamples are found over the no-cut abstract model. In such a case, we are forced to use the abstract counterexample from the min-cut model to perform abstraction and refinement. If an abstract counterexample on a min-cut model is proven to be spurious, we use the influential register heuristic presented in Section 4.3.4 to refine the abstraction.

It is desirable to include only the necessary assignments in a counterexamples. We have two model checking algorithms to achieve this goal for min-cut models that have different number of inputs. The first algorithm is based on the standard counterexample generation method used in symbolic reachability analysis [24]. This algorithm is suitable for circuits with relatively small number of inputs. First forward image computation from the initial states is performed using BDDs until the property is violated (or a fixpoint is reached and no counterexamples exist). Then backward images are calculated while restricting to the already calculated forward images, and an input cube is selected from the BDD at each time frame. We reduce the assignments in the counterexample using a smart cube selection heuristic. Given a BDD representing the set of abstract states, we pick a minimal weighted-path to BDD terminal 1 while assigning higher weights to the variables in *excluded* and *cuts*, but lower weights to the variables in *pis* and *included*. In this way, the counterexamples generated will less depend on variables in *excluded* and *cuts*, which are the potential source of conflicts when checking the counterexample on concrete model.

The second algorithm combines BDD and ATPG. When there are hundreds of inputs in the abstract model, backward image computation to generate the BDD over free inputs is very expensive. Although BDD subsetting can be used to dynamically under-approximate the BDDs, without careful control of the parameters, subsetting could generate empty BDDs, which are not useful to construct the counterexample. In the combined BDD/ATPG method, after BDD-based forward image computation is performed to hit one of the bad states, ATPG is used to generate assignments to the free inputs for each time frame. More specifically, in each backward step, we use backward image computation to obtain the pre-image (over the visible registers only, the free inputs are quantified early) of a target state, then a minimal state cube (corresponds to the shortest path to BDD terminal 1) is selected. Using this state as the initial state, and the target state as the state to reach, a combinational ATPG is used to compute the required assignments to the free inputs. As in the first method, we prefer more assignments to variables in *pis* and *included* than variables in *excluded* and *cuts*. For ATPG, this can be achieved by giving higher values of controllability to variables in *pis* and *included*, so that ATPG will try to avoid assignments to variables in *excluded* and *cuts*.

We prefer ATPG to SAT in both generating and lifting counterexamples. Although SAT has similar capacity as ATPG, for a satisfiable instance, SAT tries to make every clause true, and therefore more assignments to the model inputs are needed than ATPG, which will adversely affect the chance to find real counterexamples.

4.3.3 Check Counterexample

After a counterexample is generated from the abstract model, it is checked on the concrete model. If it is also a counterexample, then the property is violated, and the verification is done; otherwise this abstract counterexample is spurious, and error diagnosis will be performed to compute the invisible registers that need to be refined. As is presented in Section 4.3.1, we use Guided SAT/ATPG to increase the chance in finding real counterexamples.

Since this abstraction refinement algorithm targets designs with thousands of registers and more than 100K gates, SAT is used as the engine to check abstract counterexamples, instead of BDD-based engines. Although, SAT is limited by the length of the search, the restriction to a specific abstract counterexample increases its capability to search longer. In our experiments, SAT without guidance has difficulty searching for more than 20 cycles deep, while with guidance it can search more than 130 cycles. Thus it is possible for SAT based abstraction refinement algorithms to search longer counterexamples than bounded model checking based on SAT solvers. .

4.3.4 Refinement Algorithms

If the abstract counterexample is shown to be spurious, the failed counterexample is used to perform refinement. Refinement identifies a set of important registers and adds them into the current abstraction, so that the given spurious abstract counterexample is excluded from the refined abstract model. In our experiments, the immediate support of the visible registers can include more than a thousand registers. Examining them one by one is time-consuming. We have devised four refinement heuristics, so that only a small set of invisible registers are examine in order to compute refinement. First, a list of important registers is identified and ordered by their importance. Then a minimization algorithm based on Guided SAT is used to select a minimal subset from this list.

Assignments in Counterexamples

Although the direct support of the included registers is very large, usually only a small portion of them appear in the abstract counterexamples. Our first heuristic only examine the invisible registers assigned in the abstract counterexample. The intuitive reason is that these assignments are required for the validity of the counterexample on the abstract model, thus refining them will reduce the degree of approximation and thus invalidate the current counterexample on the refined model. We associate two weights with each excluded register to represent its importance.

1. One weighting method is called *frequency*, which counts the number of times that each signal appears in the given counterexample. Thus a signal is considered important if it is assigned at many time frames in the given counterexample.

2. The second method is called *reappearance*, which counts the number of times that each signal appears in all the previously generated counterexamples. A signal is considered important if it appears in many counterexamples.

3-value Simulation

The assignments to the *excluded* and *cuts* in an abstract counterexample are arbitrary, because they are primary inputs for the abstract model. In this heuristic, we use 3-value simulation on the concrete model to find out which assignments to the excluded registers are directly conflicting with the rest of the abstract counterexample, and weight an excluded register based on the number of conflicts it incurred. In the simulation, only the real primary inputs that are assigned in the abstract counterexample are given the same value as in the counterexample, the rest of the primary inputs have value X. Then 3-value simulation is calculated for the same number of time frames as the length of the abstract counterexample. Finally, the number of conflicts for each excluded registers are calculated. An excluded register is considered to have a conflict at a time frame if both abstract counterexample and 3-value simulation assign values to it at that time frame, but the values are different. In Table 4.1, possible combinations of values for excluded registers between 3-value simulation and abstract counterexamples, as well as the value to be used in the next time frame in further 3-value simulation are listed. In this table, ? represents any value, so it is a don't care. The last two rows represent situations of conflict. When a conflict happens, the value from the abstract counterexample is used to continue the 3-value simulation to reduce the number of conflicts during further simulation. 3-value simulation is very fast but conservative, so only a small number of possible conflicts are identified.

3-value simulation	**abstract counterexample**	**simulation for next time frame**
?	X	?
$\neq 1$	0	0
$\neq 0$	1	1
1	0	0
0	1	1

Table 4.1: 3-value simulation of an abstract counterexample

Influential Registers

So far the heuristics we have considered ignore the assignments to internal cut points. Such assignments are possible in an abstract counterexample over a min-cut model. As is pointed out in Section 4.3.2, when lifting the counterexample from a min-cut model to the corresponding no-cut model fails, we use min-cut counterexample in refinement. Note that, the reason for the lifting to fail is because of the loss of combinational logic after the graph min-cut procedure. In this heuristic, we collect all the assigned cuts in the abstract counterexample, and examine the combinational logic being removed by the min-cut model. Weights are assigned to the excluded registers lies in the support of these cuts. Several structural measures are taken into account during the weighting procedure, including the number of fan-ins and fan-outs to the cuts, combinational distance to a cut, etc.

Refinement Minimization

Using the heuristics described above, each excluded register is associated with four weights. For each register, a final weight is calculated by summing the given four weights together. An ordered list of the excluded registers is then calculated using these final weights. Although refining the whole list generates a model that could eliminate the counterexample, the size of the refined model could become unnecessarily large. In fact, our experience shows that usually one or two key registers are needed to disable a counterexample. Given a list of ordered candidates for refinement, an algorithm, called ***Minimize***, is used to identify a minimal subset of them, so that after this subset is refined, the counterexample is invalidated. The algorithm has two phases. The first one corresponds to the first loop, which finds a prefix of the given list of *candidates* to eliminate the *counterexample*. The second phase minimize the subset, *addedRegister*, by identifying members of it, whose removal keeps invalidating the counterexample.

```
 Algorithm Minimize (abst_model, candidates, counterexample, goal)
    addedRegister = empty;
    Foreach register in candidates do
        Add the register into abst_model
        search = GuidedSAT(abst_model, counterexample, goal)
        AddedRegister += register
    while search = reachable
    Foreach register in addedRegister do
        Remove the register from abst_model
        search = GuidedSAT(abst_model, counterexample, goal)
        if (search == unreach)
            addedRegister = addedRegister - register
        else
            Add the register into abst_model
    while TRUE
return addedRegister
```

Figure 4.3: Refinement Minimization Algorithm

4.3.5 Experimental Results

We have implemented the RFN algorithm in C. The prototype system includes a symbolic model checker implemented using the BDD package in [14], an ATPG program and a 3-value simulation program.

We performed two types of experiments on some real-world RTL designs. The first type of experiments is property verification, in which we verify that none of the target states specified by the unreachability property can be reached from an initial state. The purpose of this type of experiments is obvious — we would like to compare the property verification (and falsification) capability of RFN against plain symbolic model checking. To be fair, we perform symbolic model checking with cone-of-influence (COI) reduction.

We verified five properties against two real-world Verilog designs. The gate-level designs were obtained from logic synthesis. The first two properties "mutex" and "error flag" were verified against a module of a processor design. The next three properties "push_hf", "push_af" and "push_full" were verified against a FIFO controller design. All properties are interesting safety properties that the designers wanted to verify. Each safety property was modeled as an unreachability property with a watchdog module that asserts its output when the property is violated. In Table 4.2, the first column shows the names of the properties. The second and the third columns respectively show the number of registers and the number of gates in the COI of the properties. The fourth column shows the CPU time that RFN took to verify or falsify the properties. The fifth column shows the verification results (T=True and F=False). The last column shows the number of registers in the abstract model when RFN terminates.

We also applied our symbolic model checker to verify these properties with the COI reduction. Our symbolic model checker failed to verify any of the above five properties. Therefore, RFN enabled the formal verification of these properties that cannot be verified by our symbolic model checker. The violated property "error_flag" indicated a violation to the specification of the design. The generated error trace was 30-cycle long.

Properties	No. registers in COI	No. gates in COI	Time (sec)	Result	No. registers in abstract model

mutex	4,982	111,151	9,795	T	57
error flag	4,986	111,203	5,830	F	55
psh_hf	135	3,770	480	T	49
psh_af	135	3,771	1,075	T	42
psh_full	135	3,765	180	T	42

Table 4.2: Property Verification Results

The second type of experiments is unreachable-coverage-state analysis. The unreachable-coverage-state analysis problem is as follows. We are given a set of signals, called the coverage signals, of the gate-level design. A coverage state is a combination of the values of the coverage signals. The objective is to identify as many unreachable coverage states (on the original design, not the subcircuit containing only the coverage signals) as possible. The application of unreachable-coverage-state analysis to coverage analysis is described in [8].

RFN can be used to perform unreachable-coverage-state analysis as follows. In Step 2, we project the forward fixpoint to the set of coverage signals and identify the coverage states that are not in the projected fixpoint as unreachable. In Step 4, we mark the reached coverage states by projecting the reached states of the original design to the coverage signals. At the end of an iteration, the coverage states that have not been identified as unreachable or marked as reachable become the target states for the next iteration of RFN.

An alternative method for generating abstract models is the BFS method introduced in [8]. The BFS method relies on topological information of the gate-level design to generate abstract models. Given a size k, the BFS method first computes from the original design a min-cut subcircuit that contains the closest k registers to the coverage signals. Then it performs forward fixpoint computation on the min-cut subcircuit to identify unreachable coverage states.

The purpose of this type of experiments is to compare the quality of the abstract models generated by RFN against the quality of the abstract models generated by BFS, in terms of the number of unreachable coverage states that they identify. We performed unreachable-coverage-state analysis for seven sets of coverage signals selected from two real-world Verilog designs. The first five sets of coverage signals are selected from the Integer Unit (IU) of the Sun picoJava microprocessor [15]. The next two sets of coverage signals are selected from a USB bus controller design. Each of the first five sets of coverage signals contain 10 distinct coverage signals that introduce 1024 coverage states. The last two sets contain 6 and 21 coverage signals, respectively. The coverage signals were selected among the registers that encode control state machines.

The results of the experiments are summarized in Table 4.3. The BFS abstract models contain exactly 60 registers in each experiment. We picked the number 60 based on our experience that the forward fixpoint computation almost always completes on an abstract model with 60 registers. We applied a time limit of 1,800 CPU seconds to each RFN experiment.

In Table 4.3, the first column shows the code names of the sets of coverage signals. The second and third columns respectively show the number of registers and gates in the COIs of the coverage signals. We were a little bit surprised when we saw that the sizes of the COIs of the first five sets of coverage signals are exactly the same. The coverage signals are likely to be in a strongly connected component of the gate-level design. The fourth column shows the number of unreachable coverage states identified by RFN. The fifth column shows the number of registers in the abstract model before the time out. The sixth and seventh columns respectively show the number of unreachable coverage states identified by BFS and the time taken by BFS.

From Table 4.3 we can see that RFN uniformly beats or matches the BFS results. In addition, the time taken by BFS is more unpredictable (10,000 seconds for IU5) than RFN.

Cov. signals	No. registers in COI	No. gates in COI	No. unreach by RFN	No. registers in RFN	No. unreach by BFS	BFS time (sec)
IU1	4,458	74,258	448	40	256	5,006
IU2	4,458	74,258	736	43	256	767
IU3	4,458	74,258	880	48	880	867
IU4	4,458	74,258	448	36	256	2,667
IU5	4,458	74,258	784	42	664	10K
PE1	6,747	252,935	42	30	32	183
PE2	4,460	173,924	2,076,160	50	2,067,136	562

Table 4.3: Unreachable-coverage-state analysis results

4.4 Invisible Variables in Unsatisfiability Proofs

In this section, we introduce a proof based approach for localization reduction (PBL). In Section 4.3, candidates for refinement are based on those invisible registers that are assigned in the abstract counterexample. In PBL, the abstract counterexamples only assign values to real primary inputs and visible registers. Furthermore, SAT conflict dependency analysis is used to select the refinement candidates. We believe there are two advantages to disallowing invisible registers in the abstract counterexample. First of all, generating an abstract counterexample over invisible registers is computationally expensive, because the number of invisible registers is often large. In fact, for efficiency reasons, a BDD/ATPG hybrid engine is used in RFN to model check the abstract model. By quantifying the invisible variables early during image computation, we avoid this bottleneck. More importantly, in RFN, invisible registers are free inputs in the abstract model, their values are totally unconstrained. When checking such an abstract counterexample on the concrete machine, it is more likely to be spurious. In our case, the abstract counterexample only includes assignments to the real primary inputs and visible registers, hence a real counterexample can be found more easily.

The choice of which invisible registers to make visible is the key to the success of the refinement algorithm. Ideally, we want this set of registers to be small and still be able to prevent the spurious trace. Obviously, the set of registers appearing in the conflict graphs during the checking of the counterexample could prevent the spurious trace. However, this set can be very large. We will show here that it is unnecessary to consider all conflict graphs.

4.4.1 Identifying Important Variables

If the given abstract counterexample is spurious, refinement is done to make relevant invisible registers visible, so that the spurious abstract counterexample is invalidated in the refined model. Assume that the counterexample can be simulated up to the abstract state ce_{k-1}, but not up to ce_k. Thus formula 4.2 is satisfiable while formula 4.3 is *not* satisfiable, as shown in Figure 4.4.

$$\begin{aligned}(S_0(s_0) \wedge \gamma(ce_0)(s_0)) \quad \wedge \quad & (R(s_0, s_1) \wedge \gamma(ce_1)(s_1)) \wedge \ldots \\ & \wedge(R(s_{k-2}, s_{k-1}) \wedge \gamma(ce_{k-1})(s_{k-1}))\end{aligned} \tag{4.2}$$

$$\begin{aligned}(S_0(s_0) \wedge \gamma(ce_0)(s_0)) \quad \wedge \quad & (R(s_0, s_1) \wedge \gamma(ce_1)(s_1)) \wedge \ldots \\ & \wedge(R(s_{k-1}, s_k) \wedge \gamma(ce_k)(s_k))\end{aligned} \tag{4.3}$$

Using the terminology introduced in [23], we call the abstract state ce_{k-1} a *failure state*. The abstract state ce_{k-1} contains many concrete states given by all possible combinations of invisible variables, keeping the same values for variables as given by ce_{k-1}. The concrete states in ce_{k-1} reachable from the initial states following the spurious counterexample are called the *deadend* states. The concrete states in ce_{k-1} that have a reachable set in ce_k are called *bad* states. Because the deadend states and the bad states are part of the same abstract state, we get the spurious counterexample. The refinement

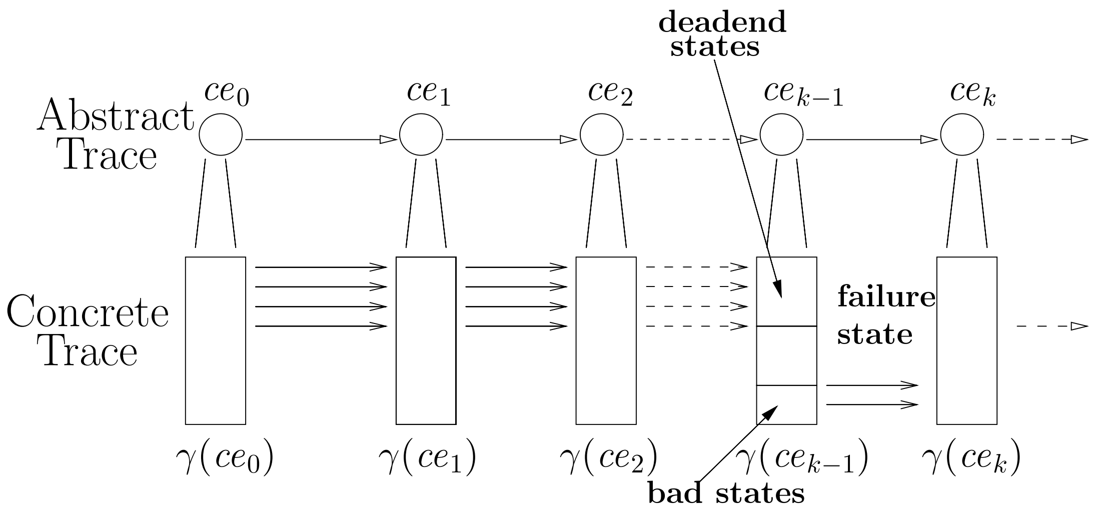

Figure 4.4: A spurious prefix and the associated deadend/bad states.

step then is to separate deadend states and bad states by making a small subset of invisible variables visible. It is easy to see that the set of deadend states are given by the values of state variables in the $(k-1)^{th}$ step for all satisfying solutions to Equation 4.2. Note that in symbolic simulation formulas, we have a copy of each state variable for each time frame.

We do this symbolic simulation using the SAT checker Chaff [56, 76]. We assume that there are concrete transitions which correspond to each abstract transition from ce_i to ce_{i+1}, where $0 \leq i < k$ (Otherwise, we use the refinement algorithms in Section 5.2.1 to remove the spurious abstract transitions). In this case, the set of *bad* states is not empty. Since $\langle ce_0, \ldots, ce_k\rangle$ is the shortest prefix that is unsatisfiable, there must be information passed through the invisible registers at time frame $k-1$ in order for the SAT solver to prove $\langle ce_0, \ldots, ce_k\rangle$ is unsatisfiable on the concrete model. The SAT solver implicitly generates constraints on the invisible registers at time frame $(k-1)$

based on both the last abstract transition and the prefix $\langle ce_0, \ldots, ce_{k-1}$. Obviously the intersection of these two constraints on those invisible registers is empty. Thus the set of invisible registers that are constrained in time frame $(k-1)$ during the SAT search is sufficient to separate *deadend* states and *bad* states (The formal proof is given in Lemma 5.2.3 of Section 5.2.2). Therefore, our algorithm limits the refinement candidates to the registers that are constrained in time frame $(k-1)$.

The proof graph records the reasons for unsatisfiability. Therefore, only the variables appearing in the proof graph are important. Instead of collecting all the variables appearing in any conflict graph, those in the proof graph are sufficient to disable the spurious counterexample. When Chaff terminates with unsatisfiability, we collect the clauses from the proof graph. Recall from Section 3.2.1, these clauses become the dependent set of the last empty conflict clause. Therefore, the set of invisible registers that, when expanded to time frame $(k-1)$, correspond to literals in these clauses are the candidates for refinement.

4.4.2 Refinement Minimization

The set of refinement candidates identified from conflict analysis is usually not minimal, i.e., not all registers in this set are required to invalidate the current spurious abstract counterexample. To remove those that are unnecessary, we have adapted the greedy refinement minimization algorithm in Section 4.3.4. The algorithm in Section 4.3.4 has two phases. The first phase is the addition phase, where a set of invisible registers that suffices to disable the spurious abstract counterexample is identified. In the second phase, a minimal subset of registers that is necessary to disable the counterexample is identified. Their algorithm tries to see whether removing a newly added register from the abstract model still disables the abstract counterexample. If that is the case, this register is unnecessary and is no longer considered for refinement. In our case, we only need the second phase of the algorithm, because the set of refinement candidates provided by our conflict dependency analysis algorithm already suffices to disable the current spurious abstract counterexample. Since the first phase of their algorithm takes at least as long as the second phase, this should speed up this minimization algorithm considerably. The Detailed description of the minimization algorithm is deferred until in Section 5.2.2 (Refer to Figure 5.1 for an illustration of the algorithm).

4.4.3 Experimental Results

We have implemented our abstraction refinement framework on top of NuSMV model checker [20]. We modified the SAT checker Chaff to produce conflict dependency graphs and to do incremental SAT. The IU-p1 benchmark was verified by conflict analysis based refinement on a SunFire 280R machine with two 750Mhz UltraSparc III CPUs and 8GB of RAM running Solaris (This is because there is a 72 long abstract counterexample for IU-p1. Checking such a counterexample using SAT by unrolling the concrete model consumes a lot of memory.). All other experiments were performed on a dual 1.5GHz Athlon machine with 3GB of RAM running Linux. We compare several verification algorithms: Cadence SMV (CSMV) which is a state of the art BDD based model checker, heuristic score based abstraction refinement [18], ILP and machine learning based abstraction refinement [22], and the proof based abstraction refinement presented in this section. In the heuristic score based abstraction refinement, all conflict graphs during the SAT search are considered. While the proof based algorithm only considers those in the unsatisfiability proof.

The experiments were performed on two sets of benchmarks. The first set of benchmarks in Table 4.4 are industrial benchmarks obtained from various sources. The benchmarks IU-p1 and IU-p2 refer to the same circuit, IU, but different properties are checked in each case. This circuit is an integer unit of a picoJava microprocessor from Sun. The D series benchmarks are from a processor design. The properties verified were simple **AG** properties. The property for IU-p2 has 7 registers, while IU-p1 and D series circuits have only one register in the property. The circuits in Table 4.5 are various abstractions of the IU circuit. The property being verified has 17 registers. They are smaller circuits that are easily handled by our methods but they have been shown to be difficult to handle by Cadence SMV [22]. We include these results here to compare our methods with the results reported in [22] for property 2. We do not report the results for property 1 in [22] because it is too trivial (all counterexamples can be found in 1 iteration). It is interesting to note that all benchmarks but IU-p1 and IU-p2 have a valid counterexample.

In Table 4.4, we compare our methods against the BDD based model checker Cadence SMV (CSMV) and heuristic score based refinement. We enabled cone of influence reduction and dynamic variable reordering in Ca-

circuit	# regs	ctrex length	CSMV time	Heuristic Score			Proof		
				time	iters	# regs	time	iters	# regs
D2	105	15	152	105	10	51	79	11	39
D5	350	32	1,192	29	3	16	38.2	8	10
D6	177	20	45,596	784	24	121	833	48	90
D18	745	28	>4 hrs	12,086	69	346	9,995	142	253
D20	562	14	>7 hrs	1,493	56	281	1,947	74	265
D24	270	10	7,850	14	1	6	8	1	4
IU-p1	4855	true	-	9,138	22	107	3,350*	13	19
IU-p2	4855	true	-	2,820	7	36	712	6	13

Table 4.4: Comparison between Cadence SMV (CSMV), heuristic score based refinement and proof based refinement for larger circuits.

dence SMV. The performance of "vanilla" NuSMV was worse than Cadence SMV, hence we do not report those numbers. We report total running time, number of iterations and the number of registers in the final abstraction. The columns labeled with "Heuristic Score" report the results with the heuristic score method. The columns labeled with "Proof" report the results of the proof based refinement. A "-" in a cell indicates that the model checker ran out of memory.

Table 4.5 compares the three localization reduction algorithms: heuristic score based, ILP and machine learning based, and proof based abstraction refinement. The results obtained using ILP and machine learning based methods is listed in column 4. in [22] for property 2.

We can see that the proof based method outperforms a standard BDD based model checker, the method reported in [22] and the heuristic score based method. We also conclude that the computational overhead of the proof based method is well justified by the smaller abstractions that it produces. The variable scoring based method does not enjoy the benefits of reduced candidate refinement sets obtained through SAT conflict dependency

circuit	# regs	ctrex length	[22] time	Heuristic Score			Proof		
				time	iters	# regs	time	iters	# regs
IU30	30	11	6.5	2.3	2	27	1.9	4	20
IU35	35	20	11	8.9	2	27	10.4	5	21
IU40	40	20	16.1	28.4	3	32	13.3	6	22
IU45	45	20	22.1	32.9	3	32	25	6	22
IU50	50	20	85.1	36	3	32	32.8	6	22
IU55	55	11	-	43	2	27	61.9	4	20
IU60	60	11	-	52.8	2	27	65.5	4	20
IU65	65	11	-	50.3	2	27	67.5	4	20
IU70	70	11	-	55.6	2	27	71.4	4	20
IU75	75	11	130.5	38.5	4	37	15.7	5	21
IU80	80	11	153.4	47.1	4	37	21.1	5	21
IU85	85	11	167.7	44.7	4	37	24.6	5	21
IU90	90	11	167.1	49.9	4	37	24.3	5	21

Table 4.5: Comparison between [22], heuristic score based refinement and proof based refinement for smaller circuits.

analysis. Therefore, it results in a bigger abstraction in general. The heuristic based refinement method adds 5 registers at a time, resulting in some uniformity in the final number of registers, especially evident in Table 4.5. Due to the smaller number of refinement steps it performs, the total time it has to spend in model checking abstract machines may be smaller (as for D5, D6, D20, IU60, IU65, IU70).

4.4.4 Performance Improvements

In this subsection, we give several algorithms to improve the localization reduction system (PBL) presented in this section.

The set of visible registers in the initial abstraction includes all the registers appearing in the given property to be proven. To make the abstraction refinement process converge more quickly, a possible way is to start with a bigger set of visible registers. Given a safety property, usually bounded model checking is first performed with a relatively small bound to see whether the given property can be easily shown to be false. This involves solving a series of satisfiability problems. When all this SAT problems are unsatisfiable, we know that the property can not be violated within the given bound. We can take advantage of BMC to identify important registers. Since the SAT instances involved are unsatisfiable, we can use the SAT conflict dependency analysis to extract the proofs of unsatisfiability. The registers appearing in these proofs are important, if we add all of them as visible registers into the initial abstraction, the abstract model can make sure that any abstract counterexample must be longer than the given BMC bound. Thus, we have speeded up the convergence of the abstraction refinement procedure. However, it is usually the case that, a large number of registers are in the unsatisfiability proofs. Having all of them in the initial abstraction makes the abstract model too large. As is pointed out early, the proof extraction method may not come up with the minimal proof. So instead of adding all the registers in the unsatisfiability proofs of the BMC instances, we score these registers based on the extract proofs. Then we only make a small number of registers with the highest scores visible. Currently, the scoring heuristic is very simple, the score of a register is the number of times it appears in the unsatisfiability proofs. It is possible to regard an unsatisfiability proof as a graph, and use the structure of the graph to design new scoring heuristics.

In counterexample based abstraction refinement, a single counterexample is considered, where each abstract state in the abstract counterexample gives values to all abstract state variables. We call such a counterexample *minterm based counterexample*. Our algorithm tries to make the generated counterexamples more general by selecting abstract states with less assignments to the abstract variables, so that each abstract state only specifies the necessary variables while leaving other variables unconstrained. We call such a counterexample *cube based counterexample*. For safety property, the counterexample is a path to a state where the property is violated. We can further generalize cube based counterexample, so that the counterexample represents all the shortest counterexample by representing the abstract states at each time frame using a BDD. We call the resulting counterexample *BDD based counterexample*. A BDD based counterexample can be generated by modifying the standard algorithm used for reachability analysis. Once the intersection of the reachable states and the unsafety states is not empty, we keep the intersection as a BDD. Then we go backwards by calculating the preimage of this BDD, then intersect it with the reachable BDD calculated during the forward phase. This process is repeated until the intersection with the set of initial states is calculated. The BDDs calculated in the backward phase represent all the shortest counterexamples. Once a BDD based abstract counterexample is calculated, the abstract refinement algorithms presented in this section can be modified to work with the BDD based counterexample. The modification is trivial, instead of a cube based abstract state, the new algorithm just need to understand a BDD over the visible variables. There are two issues to deal with when using BDD based abstract counterexamples.

- It is possible that, to represent a BDD based abstract counterexample requires too much memory. This can be avoided by BDD underapproximation during the backward counterexample generation phase.

- It is possible that, to verify whether a BDD based abstract counterexample corresponds to a concrete counterexample or not requires too much time using a SAT solver. For example, the SAT solver zChaff has spent more than an hour in order to check one BDD based abstract counterexample for one of the IU properties given in Table 4.4. To avoid this problem, we give a bound to the time that the SAT solver is allowed to run and abort the SAT search after that. We then use the cube based counterexample for the current abstraction refinement iteration.

In the refinement algorithm presented previously in this section, we try to find the shortest prefix of an abstraction counterexample that does not correspond to a real concrete path. Let k be the length of such a prefix. We restrict the refinement to the invisible registers at time $k-1$. In practice, to find the shortest unsatisfiable prefix may be time consuming for long counterexamples. For example, an abstract counterexample of length 72 has been generated for one of the IU properties in Table 4.4, where all the proper prefixes of the counterexample can be satisfied by the concrete model, yet the whole counterexample is unsatisfiable on the concrete model. We can avoid the time spent in checking the prefixes, by only checking the whole counterexample on the concrete model as in equation (4.1). When this returns unsatisfiability, we do not know the shortest unsatisfiable prefix. One way to disable the counterexample is to refine all registers appearing in this unsatisfiability proof. However, it is possible that there are too many registers in the proof. In such a case, we use the same scoring heuristic presented in the previous paragraph to evaluate the importance of invisible registers, and only refine those registers with the highest scores.

Chapter 5

SAT based Predicate Abstraction

Localization reduction, presented in Chapter 4, is a powerful technique for hardware verification. However, when the size of the required abstract model is over the capacity of BDD based model checking engines, localization reduction becomes infeasible. In this chapter, we present techniques based on predicate abstraction that can often produce small abstract models, thus enabling the verification of large scale hardware designs where localization reduction fails.

Predicate abstraction has been traditionally used for verifying infinite state systems, like software programs. For software model checking [6–8, 30, 61, 70], the use of predicate abstraction (or similar abstraction techniques) is essential because, most software systems are infinite state and the existing model checking algorithms cannot handle infinite state systems. Predicate abstraction can extract finite state abstract models, which are amenable to model checking, from infinite state systems. Since hardware systems are finite state, model checking (or simpler forms of abstraction, e.g., localization reduction [43]) has been traditionally used to verify them. Existing predicate abstraction techniques for verifying software are not efficient when applied to the verification of large scale hardware systems.

There are many proof obligations involved in predicate abstraction that require the use of decision procedures. Proof obligations can arise from equations (2.5) and (2.6) and also from determining whether an abstract counterexample is spurious or not. For software verification, these proof obligations are solved using general theorem provers. For the verification of hardware systems, which usually have compact representation in conjunctive normal form (CNF), we can use SAT solvers instead of general theorem provers. With the advancements in SAT technology, discharging the proof obligations using SAT solvers becomes much faster than using general theorem provers.

The abstract model built according to equations (2.5) and (2.6) is called the *most accurate abstract model*. Note that, in this abstract model, every abstract initial state has at least one corresponding concrete initial state, and every abstract transition has at least one corresponding concrete transition. However, to build the most accurate abstract model, there are exponential number (in the number of predicates) of implications that need to be checked in worst case. To reduce the abstraction time, in practice an *approximate abstract model* is constructed by intentionally excluding certain implications from consideration. Therefore, there are more behaviors in the approximate model than in the most accurate abstract model. We call the abstract transitions that do not have any corresponding concrete transitions *spurious transitions* (Precise definitions are given in Section 5.2.1). Since an approximate abstract model contains all the behaviors of the original concrete system, the preservation theorem still holds, which guarantees the correctness of a universal temporal logic formula on the concrete system once it has been proven on the abstract model. To further reduce the abstraction time, we present a technique to reduce the number of implications that are checked in building the abstract models using equations (2.5) and (2.6).

There are two cases for an abstract counterexample to be spurious: One is that there is a spurious transition, that is, an abstract transition which does not have any corresponding concrete transitions; the other is that the counterexample has a spurious prefix, that is, there are no concrete paths that correspond to the prefix.

Our first SAT based algorithm deals with the first case. Recall that, it is time consuming to build the most accurate abstract model when the number of predicates is large. Since it is usually the case that, the most accurate abstract model is not necessary in order to verify the given property, we compute an approximate abstract model initially and rely on refinement to make the abstract model as accurate as possible. We use a heuristic similar to the one given in [6] to build an approximate abstract model. Instead of considering all possible implications of the form $\hat{Y} \rightarrow \hat{Y}'$ we impose restriction on the lengths of $\hat{Y}$ and $\hat{Y}'$ in equation (2.6) (The approximation to the set of abstract initial states can be similarly done for equation (2.5)). If the resulting abstract model is too coarse, an abstract counterexample with a spurious transition might be generated. This spurious transition can be removed by adding an appropriate constraint to the abstract model (details are given in Section 5.2.1). The constraint should be made as general as possible so that many related spurious transitions are also removed. An algorithm for this has been proposed in [28] which in the worst case requires $2m$ number of calls to a theorem prover, where m is the number of predicates. We propose a new algorithm, based on SAT conflict dependency analysis (presented in Section 3.2), to generate a general constraint without any additional calls to the SAT solver. Our algorithm works by analyzing the conflict graphs generated when detecting the spurious transition. Thus our algorithm can be much more efficient than the algorithm in [28].

Even after removing spurious transitions there could be a spurious prefix of the given abstract counterexample. This happens because the set of predicates is not enough to capture the relevant behaviors of the concrete system. In such a case, a new predicate is identified and added to the current abstract model to invalidate the counterexample. To make the abstraction refinement process efficient, it is desirable to compute a predicate that can be compactly represented. Large predicates are difficult to compute and discharging any proof obligation involving them will be slow. We propose an algorithm, again based on SAT conflict dependency analysis, to reduce the number of concrete state variables that the new predicate depends on. Then the predicate is calculated by a projection-based SAT enumeration algorithm. Experiments show that this algorithm can efficiently compute the required predicates for design with thousands of registers.

5.1 SAT based Abstraction

In this section, we describe a SAT based algorithm to build the abstract model. Since the concrete transition relation must be used during abstraction, it is not feasible to use BDD based existential quantification [23] for large designs. Traditional predicate abstraction algorithms [6, 66] abstract each line of code separately, or for improved accuracy, each basic block separately. It is well known [47] that pushing abstraction to each line of code (or each basic block) results in an over-approximation. Big Verilog designs can have more than quarter million lines of code. So line by line abstraction is infeasible for such designs. Since variables in hardware designs are usually bit-vectors with small length, and the predicates involved in hardware verification are propositional formulas, we abstract the circuit as a whole instead of abstracting it part by part.

Recall from Section 2.3 that, $\hat{R}$ is given by the equation

$$\hat{R} = \bigwedge\{\hat{Y} \rightarrow \hat{Y'} \mid (R \wedge \gamma(\hat{Y})) \rightarrow \gamma(\hat{Y'})\}.$$

where $\hat{Y}$ is a conjunction of literals over the current state abstract variables B_is and $\hat{Y'}$ is a disjunction of literals over the next state abstract variables B_i's (from now on we call each pair consisting of $\hat{Y}$ and $\hat{Y'}$ used in building $\hat{R}$ a *testpoint*). As discussed earlier this equation is equivalent to the standard existential formulation:

$$\hat{R}(\hat{s_1}, \hat{s_2}) = \exists s_1 s_2 . \rho(s_1, \hat{s_1}) \wedge \rho(s_2, \hat{s_2}) \wedge R(s_1, s_2).$$

In order to build the abstract transition relation $\hat{R}$, we must consider all possible pairs $\hat{Y}$ and $\hat{Y'}$. For each such pair we need to determine whether the implication

$$(R \wedge \gamma(\hat{Y})) \rightarrow \gamma(\hat{Y'})$$

is a tautology or not. To do that we can negate the formula, translate it into CNF and give it to a SAT solver. If the result is unsatisfiable, then the implication is a tautology. Note that R is common to all the formulas given to the SAT solver. This means the incremental SAT technique described in Section 3.2.2 can be used, so that the conflict clauses derived from R alone are generated only once.

After having constructed $\hat{R}$, we need to construct the set of abstract initial states. This can be done with SAT solvers as well. Recall that the set of abstract initial states is given by

$$\hat{S_0} = \bigwedge \{\hat{Y} \mid S_0 \rightarrow \gamma(\hat{Y})\}.$$

where S_0 is the set of initial states and $\hat{Y}$ is a disjunction of current state abstract variables B_is. For each $\hat{Y}$ we can convert the negation of the condition $S_0 \rightarrow \gamma(\hat{Y})$ into CNF and give it to SAT solver. If the CNF formula is unsatisfiable then the condition $S_0 \rightarrow \gamma(\hat{Y})$ is a tautology.

5.1.1 Reducing the number of testpoints

In building the exact abstract transition relation we need to consider all testpoints $(\hat{Y}, \hat{Y'})$. The number of different testpoints is exponential in the number of abstract state variables. Experiments show that the efficiency of abstraction is mainly dominated by the number of calls to the SAT solver. So we restrict the number of literals in $\hat{Y}$ to be less than or equal to 2, and we only allow one literal in $\hat{Y'}$. Because of this restriction, our abstraction will be an over-approximation of the exact transition relation. An inexact transition relation can lead to spurious counterexamples, but it is better to go ahead with an approximate abstract model rather than to spend considerable effort building the exact abstract system. Experiments demonstrate that spending too much effort building the abstract model can hurt the overall runtime because an accurate abstract model may not really be necessary. Given m predicates, the total number of the restricted testpoints is $O(m^3)$ using our length restriction.

Because the number of testpoints is $O(m^3)$, where m is the number of abstract state variables (which is the same as the number of predicates), building the abstract model can still be too expensive in practice. When a testpoint is determined to be a tautology, it may make some other testpoints trivially true or false (See [66] for details.). In practice, the application of this heuristic is limited, because most of the testpoints are not tautologies. To reduce the number of calls to the SAT solver, which is the main bottleneck, we try to check many testpoints in a single call to the SAT solver if possible.

Given two testpoints $t^1 = \hat{Y^1} \rightarrow \hat{Y^{1\prime}}$ and $t^2 = \hat{Y^2} \rightarrow \hat{Y^{2\prime}}$, the *combined testpoint* is $\hat{Y^1} \wedge \hat{Y^2} \rightarrow \hat{Y^{1\prime}} \vee \hat{Y^{2\prime}}$. Two testpoints t^1 and t^2 are *compatible* iff $\hat{Y^1} \wedge \hat{Y^2} \neq false$ and $\hat{Y^{1\prime}} \vee \hat{Y^{2\prime}} \neq true$. We call the combined testpoint of two incompatible testpoints a *trivial testpoint.* To speed up the process, we use the following simple heuristic to decide whether two testpoints are compatible or not. The conjunction of $\hat{Y^1}$ and $\hat{Y^2}$ is $false$ if the conjunction contains both B_i and $\neg B_i$ (Similarly for $\hat{Y^{1\prime}}$ and $\hat{Y^{2\prime}}$). If the combined testpoint is not a tautology, then neither t^1 nor t^2 is tautology. This fact can be used to reduce the number of calls to the SAT solver in building the abstract model. Given a list of testpoints, we first partition them, so that the combined testpoint for each partition is not trivial. For a partition, we first test whether the combined testpoint is a tautology or not. If it is, we add it to the abstract model and then check each testpoint in the partition as usual. If the combined testpoint is not a tautology, then none of the testpoints in the partition is a tautology and all testpoints in this partition can be excluded from further consideration. By this method we can effectively reduce the number of calls to the SAT solver.

5.2 SAT based Refinement

We first introduce some notation to represent the unrolling of a transition system from initial states. Let V be a set of variables, let the corresponding set of next state variables be V'. We call V and V' *untimed variables*. For every variable in V we maintain a version of that variable at each time $i \geq 0$. If V is a set of state variables, then V^i, is the set of timed versions of variables in V at time $i \geq 0$. We call V^i *timed variables* at time i. Using timed abstract state variables B^i corresponding to a set of abstract state variables B, an *abstract counterexample* $ce(B^0, \ldots, B^n)$ is a sequence of abstract states $\langle ce_0(B^0), ce_1(B^1), \ldots, ce_n(B^n)\rangle$, where $ce_i(B^i)$ is a cube over the abstract variables at time i. When it is clear from context, we sometimes represent a counterexample without explicitly mentioning timed variables. Let $f(V)$ be a boolean function, which maps the set of states over variables V to $\{0, 1\}$. The *timed* version of f at time i, denoted by $f^i(V^i)$, is the same function as f except that it is over the timed variables V^i. We define an operator, called *utf* (for untimed function), which for a given timed function $f^i(V^i)$, returns the untimed function $f(V)$, i.e, $f(V) = \mathit{utf}(f^i(V^i))$. Given a relation $r(V, V^{'})$, which maps the set of states over current state variables V to the set of states over the next state variables V', $r^i(V^i, V^{i+1})$ is the *timed* version of r at time i. We define an operator, called *utr* (for untimed relation), which for a given timed relation $r^i(V^i, V^{i+1})$, returns the untimed relation $r(V, V^{'})$,i.e., $\mathit{utr}(r^i(V^i, V^{i+1})) = r(V, V^{'})$.

Let $B = \{B_1, \ldots, B_m\}$ and V be the set of abstract and concrete state variables, respectively. Given a timed abstract expression f in terms of B^i at time i, its concretization is a timed concrete expression $\gamma(f)$ in terms of V^i obtained by replacing each B^i_j in f with the timed version of the corresponding predicate P^i_j. Let $ce = \langle ce_0, ce_1, \ldots, ce_n\rangle$ be an abstract counterexample. Let i be a natural number, such that $0 < i \leq n$. The set of pairs of concrete states corresponding to the abstract transition from ce_{i-1} to ce_i is

$$\mathit{trans}(i-1, i) = \gamma(ce_{i-1}) \wedge R^{i-1} \wedge \gamma(ce_i) \tag{5.1}$$

The set of concrete paths which corresponds to the prefix of the abstract counterexample up to time i, is a set of lists of concrete states $\{\langle s_0(V^0), \ldots, s_i(V^i)\rangle\}$

that satisfy the following equation:

$$prf(i) = S_0 \wedge \gamma(ce_0) \wedge R^0 \wedge \cdots \wedge \gamma(ce_{i-1}) \wedge R^{i-1} \wedge \gamma(ce_i). \qquad (5.2)$$

Let BV be a set of boolean variables and let $BV_1 \subseteq BV$. If f is a CNF formula over BV, the *satisfiable set* of f over BV_1, denoted by $SA[BV_1](f)$, is the set of all satisfying assignments of f projected on to BV_1. Thus, $SA[BV_1](f) = proj[BV_1](SA[BV](f))$. For a SAT solver with conflict based learning, there is a well known algorithm to compute $SA[BV_1](f)$ without first computing $SA[BV](f)$ [49]. Once a satisfiable solution is found, a blocking clause over BV_1 is created to avoid generating the same projected solution. After this blocking clause is added, the SAT search continues. This process repeats until the SAT solver concludes that the set of clauses is unsatisfiable, i.e., there are no further solutions. The set of all satisfying assignments over BV_1 is the required result, which can be represented as a DNF formula.

Given a set of variables SV that are not necessarily boolean, let BSV be the set of boolean variables in the boolean encoding of variables in SV. Let f be a CNF formula over BSV.The *scalar support of the CNF formula* f, denoted by $ssuppt[SV](f)$, is a subset of SV that includes a variable $v \in SV$ iff at least one of v's corresponding boolean variables is in f.

An abstract counterexample $ce = \langle ce_0, ce_1, \ldots, ce_n \rangle$ is a real counterexample if and only if the set $prf(n)$ is not empty. If the abstract counterexample is a real counterexample, then the property is false on the concrete machine. Otherwise the counterexample is spurious and we need to refine the current abstract model. There are two possible reasons for the existence of a spurious counterexample: One is that the computed abstract model is an over-approximation of the most accurate abstract model. The other is that the set of predicates is insufficient to model the relevant behaviors of the system. In Section 5.2.1, we describe how our algorithm deals with the first case (we only show how to remove spurious transitions from the abstract transition relation. The refinement for an approximate set of abstract initial states is similar.). In Section 5.2.2 we deal with the case where the the set of predicates is not sufficient.

5.2.1 Refinement to Exclude Spurious Transitions

Given an abstract counterexample $ce = \langle ce_0, ce_1, \ldots, ce_n \rangle$, if there exists i, $0 < i \leq n$, such that the set $\mathit{trans}(i-1, i) = R^{i-1} \wedge \gamma(ce_{i-1}) \wedge \gamma(ce_i)$ is empty, then we call the transition from ce_{i-1} to ce_i a *spurious transition*. That is, there are no concrete transitions corresponding to the abstract transition from ce_{i-1} to ce_i. Clearly, the counterexample is not a real counterexample. To determine whether $\mathit{trans}(i-1, i)$ is empty or not, we convert it into a SAT unsatisfiability problem. Since, in the most accurate abstract model, there is at least one concrete transition corresponding to every abstract transition, spurious transitions exist only for approximate abstract transition relations.

Since spurious transitions are not due to the lack of predicates but due to an approximate abstract transition relation, our algorithm removes spurious transitions by adding appropriate constraints to $\hat{R}$. For the spurious transition from ce_{i-1} to ce_i, we have $R^{i-1} \wedge \gamma(ce_{i-1}) \wedge \gamma(ce_i) \Leftrightarrow$ false. Therefore, $R^{i-1} \Rightarrow (\gamma(ce_{i-1}) \rightarrow \gamma(\neg ce_i))$. Note that ce_{i-1} is a conjunction over the abstract state variables at time $i-1$, and $\neg ce_i$ is a disjunction over the abstract state variables at time i. Since the concrete transition relation does not allow any transition from $\gamma(ce_{i-1})$ to $\gamma(ce_i)$ we should add the constraint $utr(ce_{i-1} \rightarrow \neg ce_i)$ to $\hat{R}$. The resulting transition relation is correct and disallows the spurious transition. The constraint $ce_{i-1} \rightarrow \neg ce_i$ can potentially involve most of the abstract state variables, thus making it very specific and not useful in general. It is advantageous to make the constraint as general as possible (thus making the abstract transition relation more accurate), provided that the cost of achieving this is not too large. In the rest of this subsection, we describe an efficient algorithm which removes some of the literals from ce_{i-1} and ce_i in $ce_{i-1} \rightarrow \neg ce_i$, making the constraint more general.

Computing A General Constraint. Let m be the number of predicates. The problem of finding a general constraint to eliminate a spurious transition can be formalized as follows: Given propositional formulas f and f_j where $1 \leq j \leq 2m$, which make $f \wedge \bigwedge_{1 \leq j \leq 2m} f_j$ unsatisfiable, find a small subset $care \subseteq \{1, \ldots, 2m\}$, such that $f \wedge \bigwedge_{j \in care} f_j$ is unsatisfiable. It is easy to see that if we let $f = R^{i-1}$ and let each f_j correspond to the concretization of a literal in ce_{i-1} or ce_i, then we can drop those literals that are not in *care* from $ce_{i-1} \rightarrow \neg ce_i$. The resulting constraint will be made more general. The set *care* can be efficiently calculated using the conflict dependency analysis algorithm described in Section 3.2.

Before we run the SAT solver we need to convert $f \wedge f_1 \wedge f_2 \wedge \cdots \wedge f_{2m}$ to CNF, and in this process some of the f_j's might be split into smaller formulas. Hence it may not be possible to keep track of all f_j's. To overcome this difficulty, we introduce a new boolean variable t_j for each f_j in the formula and convert the formula into

$$F = \exists t_1, t_2 \ldots, t_{2m}.\ f \wedge \bigwedge_{j \in \{1,\ldots,2m\}} (t_j \wedge (t_j \equiv f_j)). \tag{5.3}$$

It is easy to see that this formula is unsatisfiable iff the original formula is unsatisfiable, because the two are logically equivalent. Once (5.3) is translated to a CNF formula, for each t_j there is a clause T_j containing only one literal, t_j. So, instead of keeping track of f_j's directly we keep track of T_j's. Since the CNF formula F corresponding to (5.3) is unsatisfiable, we know that $SUB(F) \subseteq F$ is unsatisfiable, where $SUB(F)$ is defined as in Section 3.2. It can be shown that $\mathit{care} = \{j \mid T_j \in SUB(F)\}$ represents the desired set of f_j's. Using the set care, we can add a more general constraint to $\hat{R}$.

Lemma 5.2.1 *Let F be a unsatisfiable CNF formula as defined in equation (5.3). Let $\mathit{care} = \{j \mid T_j \in SUB(F)\}$. Then $f \wedge \bigwedge_{j \in \mathit{care}} f_j$ is unsatisfiable.*

Proof: It is easy to see the following:

F is unsatisfiable

$\Rightarrow$ $F_1 = \exists t_1, \ldots, t_{2m}.\ f \wedge \bigwedge_{j \in care}(t_j \wedge (tj \equiv f_j)) \wedge \bigwedge_{j \notin care}(t_j \equiv f_j)$ is unsatisfiable

because $\{F_1\} \supseteq SUB(F)$

$\Rightarrow$ $f \wedge \bigwedge_{j \in care} \exists t_j\ (t_j \wedge (tj \equiv f_j)) \wedge \bigwedge_{j \notin care} \exists t_j\ (t_j \equiv f_j)$ is unsatisfiable

because t_j is unique

$\Rightarrow$ $f \wedge \bigwedge_{j \in care} f_j$ is unsatisfiable

∎

It is easy to see that our algorithm only analyzes the search process of the SAT problem during which the spurious transition was identified. In [28], a potentially more general constraint than the one computed by the above algorithm can be found. It works by testing whether each f_j can be removed to keep the resulting formula unsatisfiable. Their algorithm requires $2m$ calls to a theorem prover, which is time consuming when the number of predicates, m, is large. As presented in Section 3.2, the unsatisfiable subset $SUB(F)$ may not be a minimal unsatisfiable subset of F. Consequently, in general, the set *care* our algorithm computes is not minimal. However, in practice, its size is comparable to a minimal set. It is easy to modify our algorithm to make *care* minimal. After the set *care* is computed, we can try to eliminate the remaining literals one by one as in [28], which requires $|care|$ additional calls to the SAT solver. Since the size of *care* is already small, this is not very expensive.

5.2.2 Refinement by adding a New Predicate

Even after we have ensured that there are no spurious transitions (and $\gamma(ce_0) \wedge S_0 \neq \emptyset$) in the counterexample ce, the counterexample itself can still be spurious. Let n be the length of the given abstract counterexample. We are interested in k such that $1 < k \leq n$ and the prefix $p_{k-1} = \langle ce_0, ce_1, \ldots, ce_{k-1} \rangle$ of the counterexample corresponds to a valid path but $p_k = \langle ce_0, ce_1, \ldots, ce_k \rangle$ does not. Formally, we call p_k a *spurious prefix* if and only if $\mathit{prf}(k-1) \neq \emptyset \wedge \mathit{prf}(k) = \emptyset$. If there is no such k then the counterexample is real. Otherwise, the set of states $\mathit{SA}[V^{k-1}](\mathit{prf}(k-1))$ is called the set of *deadend states*, denoted by *deadend* [23]. Deadend states are those states in $\gamma(ce_{k-1})$ that can be reached but do not have any transition to $\gamma(ce_k)$. The set of states $\mathit{SA}[V^{k-1}](\mathit{trans}(k-1,k))$ is called the set of *bad states*, denoted by *bad* [23]. The states in *bad* are those states in $\gamma(ce_{k-1})$ that have a transition to some state in $\gamma(ce_k)$.

Lemma 5.2.2 *For a spurious abstract counterexample ce without spurious transitions, let k be the length of the spurious prefix of ce. Then $\mathit{deadend} \neq \emptyset$, $\mathit{bad} \neq \emptyset$ and $(\mathit{deadend} \cap \mathit{bad}) = \emptyset$.*

Proof: Since there are no spurious transitions in ce, for every i, $0 < i \leq n$, $\mathit{trans}(i-1,i)$ is satisfiable. Thus $\mathit{bad} = \mathit{SA}[V^{k-1}](\mathit{trans}(k-1,k))$ is not empty. Since $\mathit{prf}(k-1)$ is satisfiable, the set of deadend states is not empty either. Finally, if $(\mathit{deadend} \cap \mathit{bad}) \neq \emptyset$ then $\mathit{prf}(k)$ would be satisfiable. So $(\mathit{deadend} \cap \mathit{bad}) = \emptyset$ ∎

As is pointed out in [23], it is impossible to distinguish between *deadend* and *bad* states using the existing set of predicates, because the abstraction of the two is the same abstract state ce_{k-1}. Therefore, our refinement algorithm aims to find a *separating predicate*, *sep*, such that $deadend \subseteq sep$ and $sep \cap bad = \emptyset$ (the alternative definition for *sep*, which satisfies $bad \subseteq sep \wedge deadend \cap sep = \emptyset$, also works). After introducing *sep* as a new predicate, the abstract model will be able to distinguish between the deadend and bad states. We call the set of concrete state variables over which a predicate is defined the *support* of the predicate. Our algorithm first identifies a minimal set of concrete state variables. Then a predicate over these variables that can separate the deadend and bad states is computed.

Minimizing the Support of the Separating Predicate

An important goal of our refinement algorithm is to compute a predicate that can be represented compactly (called *compact predicates* for short). For large scale hardware designs, existing refinement algorithms, such as weakest precondition calculation, preimage computation, syntactical transformation etc., may fail because the predicates they are trying to compute are too big to be represented. Our algorithm avoids this problem by first computing a minimal set of concrete state variables that are responsible for the failure of the spurious prefix. Our algorithm guarantees that there is a separating predicate over this minimal set that can separate the deadend and bad states. It is usually the case that the size of any representation of a predicate can be bound by the size of its support. Therefore, our algorithm can compute compact predicates with minimal supports.

Our algorithm to compute the desired support is similar to the one used in finding the important registers in localization reduction as described in Section 4.4 and in [18]. Since the CNF formula for $\mathit{prf}(k)$ is unsatisfiable, we can use conflict dependency analysis from Section 3.2 to identify $SUB(\mathit{prf}(k))$ that is unsatisfiable. Let all the concrete state variables at time $k-1$ whose CNF variables are in $SUB(\mathit{prf}(k))$ be $\mu(ce, k-1)$. That is $\mu(ce, k-1) = \mathit{ssuppt}[V^{k-1}](SUB(\mathit{prf}(k)))$. For the sake of brevity we will refer to $\mu(ce, k-1)$ as μ. Let $\mathit{deadend}_\mu = \mathit{proj}[\mu](\mathit{deadend})$ be the projection of the deadend states on μ. Let $\mathit{bad}_\mu = \mathit{proj}[\mu](\mathit{bad})$ be the projection of the deadend states on μ. We will show that

$$\mu \neq \emptyset \wedge \mathit{deadend}_\mu \cap \mathit{bad}_\mu = \emptyset. \tag{5.4}$$

Thus any concrete set of states S_1 that satisfies $(S_1 \supseteq \mathit{deadend}_\mu) \wedge (S_1 \cap \mathit{bad}_\mu = \emptyset)$ is a candidate separating predicate. We prove equation (5.4) through the following two lemmas:

Theorem 5.2.1 For a spurious abstract counterexample ce without spurious transitions, let k be the length of the spurious prefix. Then $\mu(ce, k-1)$ is not empty.

Proof: Since $\mathit{prf}(k) = \mathit{prf}(k-1) \wedge \mathit{trans}(k-1, k)$ is unsatisfiable, and both $\mathit{prf}(k-1)$ and $\mathit{trans}(k-1, k)$ are satisfiable. The intersection of the CNF variables of $\mathit{prf}(k-1)$ and $\mathit{trans}(k-1, k)$ corresponds to the state variables at time $k-1$, i.e, V^{k-1} is their common scalar support. Thus, $SUB(\mathit{prf}(k))$ must include some CNF variables which correspond to variables in V^{k-1}.

Lemma 5.2.3 *The intersection of $deadend_\mu$ and bad_μ is empty.*

Proof: Note that $prf(k) = prf(k-1) \wedge trans(k-1,k)$ is unsatisfiable and both $prf(k-1)$ and $trans(k-1,k)$ are satisfiable. Thus both $f_1 = \{SUB(prf(k))\} \cap \{prf(k-1)\}$ and $f_2 = \{SUB(prf(k))\} \cap \{trans(k-1,k)\}$ are satisfiable. The conjunction of $f_1 \wedge f_2$ is $SUB(prf(k))$, which is unsatisfiable. Since variables in μ are the only common variables between f_1 and f_2, and $f_1 \wedge f_2$ is unsatisfiable we have

$$SA[\mu](f_1) \cap SA[\mu](f_2) = \emptyset. \tag{5.5}$$

We also have

$$\begin{aligned} SA[\mu](f_1) &\supseteq SA[\mu](prf(k-1)) \\ SA[\mu](f_2) &\supseteq SA[\mu](trans(k-1,k)) \end{aligned}$$

So using equation (5.5), we have $SA[\mu](prf(k-1)) \cap SA[\mu](trans(k-1,k)) = \emptyset$, i.e., $deadend_\mu \cap bad_\mu = \emptyset$. ∎

To further reduce the size of μ and to make it minimal we have developed a refinement minimization algorithm, which eliminates any unnecessary variables in μ while ensuring that equation (5.4) still holds. The algorithm is illustrated in Figure 5.1. In the figure, the concrete transition relation is unrolled at time $0, 1, 2, \ldots, k-2, k$ (not at $k-1$). The gray box represents the state variables of the concrete model. For all $i < k-2$, next state variable of time i is the same as the current state variable of time $i+1$. The unrolling of the transition relation is also conjuncted with the counterexample from ce_0 to ce_{k-1}. Note that, the current state variable at time k is constrained by the concretization of the counterexample at time $k-1$, and the next state

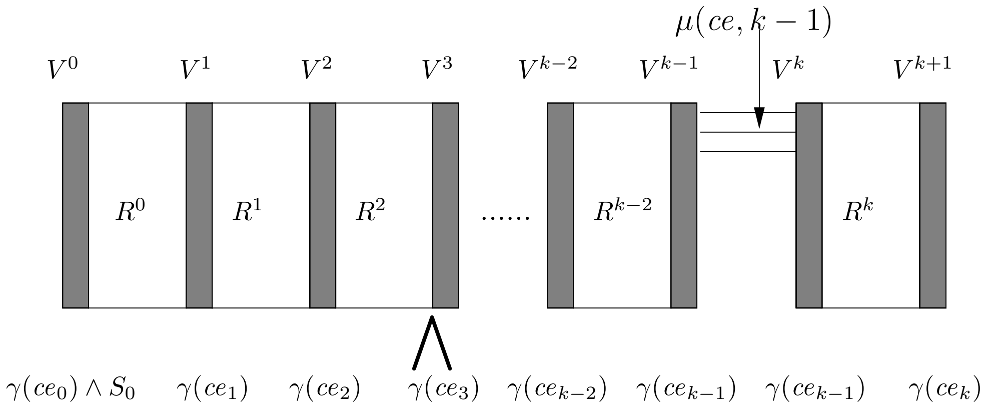

Figure 5.1: Greedy Minimization Based on Incremental SAT.

variables at time k is constrained by the concretization of the counterexample at time k. This is used to duplicate the concretization of ce_{k-1}, so that their supporting variables are disjoint. We reduce the set of state variables in $\mu(ce, k-1)$ as follows. Our algorithm starts by equating all variables in $\mu(ce, k-1)$ to the corresponding current state variable at time k. So that the resulting CNF formula should be logically equivalent to $SUB(prf(k))$, so it should be unsatisfiable. Let $\mu' = \emptyset$. For each variable $v^{k-1} \in \mu(ce, k-1)$, we remove the constraint that makes $v^{k-1} = v^k$. This new CNF formula is solved. If it is satisfiable, we add v^{k-1} into μ'. After this is repeat for each variable in $\mu(ce, k-1)$, we get a reduced set μ', which is minimal to keep $deadend_{\mu'} \cap bad_{\mu'} = \emptyset$. The proof for the correctness of this procedure can be similarly constructed according to the proof in Lemma 5.2.3. This minimization algorithm requires $|\mu(ce, k-1)|$ number of calls to the SAT solver. Since the difference between any two CNF formulas during two consecutive calls to the SAT solver is only one clause, we can use incremental SAT from Section 3.2.2 to solve them efficiently. The size of the achieved minimal set is small. In most of our experiments, the size of μ was less than 20, which is several orders of magnitude less than the total number of concrete state variables.

Computing Separating Predicates using SAT

Note that, any set of concrete states that separates $deadend_\mu$ and bad_μ is a desired separating predicate. We propose a new *projection based SAT enumeration algorithm* to compute such a separating set, which can be represented efficiently as a CNF formula or a conjunction of DNF formulas. Our algorithm has three steps. First, we try to compute bad_μ using a SAT enumeration algorithm, which avoids computing *bad* by creating blocking clauses over μ. Since the size of μ is pretty small, this procedure can often terminate quickly. If that is the case, our algorithm terminates and $\neg bad_\mu$ is the required separating predicate, which is represented as a CNF formula. Otherwise, we try to compute $deadend_\mu$ using a similar method. If this procedure finishes in a reasonably short amount of time, our algorithm terminates and $deadend_\mu$ is the desired separating predicate, which is represented as a DNF formula.

In the third case when both $deadend_\mu$ and bad_μ can not be computed within a given time limit, we compute an over-approximation of $deadend_\mu$, denoted by ODE. It is possible that the set ODE overlaps with bad_μ. Let $SODE = proj[\mu](ODE \wedge bad)$ be the intersection of the two. Then the desired separating predicate is $ODE \wedge \neg SODE$, which is represented as a conjunction of DNF formulas. In most cases, $SODE$ is much smaller than bad_μ, so it can often be enumerated using SAT. If in a rare case, even $SODE$ can not be efficiently enumerated using SAT (we do not encounter this problem for all our experiments.), we use other methods to compute a new predicate. For example, any register in μ can be added as a new predicate to make sure the abstract model is refined. We now present a projection based method to compute an over-approximation of $deadend_\mu$. We partition the variables in μ into smaller sets $\mu_1, \dots, \mu_l$ based on the closeness of the variables (the criterion for closeness is based on circuit structure [19]). Because each set is small, we can compute each $deadend_{\mu_i}$ easily. The over-approximation is $ODE = \wedge_{i \in \{1,\dots,l\}} deadend_{\mu_i}$.

After the calculated separating predicate *sep* is added as a new predicate, suppose we introduce B_{m+1} as the corresponding abstract boolean variable. Then we add the constraint $B_{m+1} \rightarrow utr(ce_{k-1} \rightarrow \neg ce_k)$ to the abstract transition relation. The following theorem says that the concrete transition relation implies the concretization of this constraint.

Theorem 5.2.2 $R \Rightarrow \gamma(B_{m+1} \rightarrow \neg(utr(ce_{k-1} \wedge ce_k)))$.

Proof: The abstract state variable B_{m+1} corresponds to a separating predicate *sep* which satisfies $sep \cap bad = \emptyset$. Thus $sep \wedge R^{k-1} \wedge \gamma(ce_{k-1}) \wedge \gamma(ce_k) \equiv$ false. Therefore, $R \Rightarrow \gamma(B_{m+1} \rightarrow \neg(utr(ce_{k-1} \wedge ce_k)))$. ∎

By adding a small predicate *sep* over the variables in μ we can eliminate the spurious counterexample *ce* and refine the abstract model $\hat{R}$. We can also remove unnecessary assignments in ce_{k-1}, ce_k to make the new constraint $B_{m+1} \rightarrow \neg(utr(ce_{k-1} \wedge ce_k))$ more general. This can be done using the method from Section 5.2.1.

5.3 Exploit RTL Information

Current predicate abstraction methods do not make use of information available in the high level descriptions of the system under verification. Most hardware design tools use high level design languages, such as ESTEREL, graphical FSMs, RTL Verilog/VHDL etc. But most model checking engines and existing verification tools use the bit level representation of the design under verification. There is much useful information that is relevant to verification in the high level representation, which is lost once the design is translated to bit level representation. To retain this information, we extract the branch conditions in RTL Verilog (the language considered in this paper) and use them as predicates. This technique can be easily adapted to other design languages.

For a given design, there are usually many branch conditions that we can extract. Not all of them are relevant to the verification of a given property. We propose a lazy counterexample based refinement algorithm to efficiently identify the branch conditions that are relevant.

5.3.1 Extracting Branch Conditions

High level design languages usually contain branch statements, such as **if**, **case** statements. The **if** statement has two branches, while the **case** statement may have multiple branches. Usually, a **case** statement can be converted to multiple **if-then-else** statements that are equivalent to it. We call the boolean predicates that determine which branch to be executed, *branch conditions*. We intend to extract the branch conditions and use them as predicates in predicate abstraction.

For the purpose of model checking, the high level representation of the system under verification is translated into a formula over the current and next state variables (referred to as the transition relation). Each extracted branch condition is translated into a subformula of the transition relation. For a branch condition, the corresponding subformula of the transition relation is called the *flattened branch condition.* The transition relation is further converted into different representations that are suitable for different model checking engines. For example, it is converted to BDDs for BDD-based model checkers, or CNF for SAT-based model checkers. For a flattened branch condition, it is straightforward to identify the corresponding representation inside the model checking engines.

We will describe a simple method to extract a set of flattened branch conditions for RTL Verilog designs. We believe it is easy to generalize this method to other design languages. One possible method is to develop a translator from RTL Verilog to gate level circuits, which can then be easily converted into a transition relation. The main disadvantage of this method is the amount of work involved in handling the semantics of Verilog, which is not formally defined [34]. In practice Verilog is interpreted by a set of standard commercial tools, such as *Synopsys Design Compiler* [41]. Our method relies on the fact that commercial synthesis tools already exist for Verilog. We first convert the RTL design into another equivalent design, where the relevant branch conditions are renamed to signals with unique names. An example is shown in Figure 5.2. We use the *continuous assignment statement* in Verilog to rename the branch conditions using unique signals, such that

```
Original design
always @(posedge clk) begin
        if (mode != NO_CONF) begin
                ...
        end else if (a == b) begin
                ...
        end
end

Modified design
assign pred1 = mode != NO_CONF;
assign pred2 = a == b;
always @(posedge clk) begin
        if (pred1) begin
                ...
        end else if (pred2) begin
                ...
        end
end
```

Figure 5.2: Replace branch conditions using unique signals

the modified design is equivalent to the original one. After this, a gate level circuit is generated from the modified design using Synopsys Design Compiler. We further translate the gate level circuit into a transition relation and the flattened branch conditions can be identified using the unique signal names. Our method can be easily applied to other design languages as long as there are language constructs to rename boolean predicates using new variables. Our method can take advantage of existing translators, therefore the implementation time is much shorter than building a translator from scratch.

It is usually the case that there are many branch conditions that we can extracted from a high level representation of designs. Not all of them can be used as predicates to build the initial abstraction, otherwise the abstract model will become too large. We use the refinement algorithm in Section 5.3.2 to identify a subset of the branch conditions which are necessary to invalidate the given spurious abstract counterexample.

5.3.2 Counterexample-based Lazy Refinement

In counterexample guided abstraction refinement, a given spurious abstract counterexample is invalidated during refinement through the introduction of a set of predicates, called *invalidating predicates*, into the abstract model. Once an abstract counterexample is determined to be spurious, the algorithm described in this subsection identifies a subset of the flattened branch conditions as invalidating predicates.

We first introduce some notation. Let f be a boolean formula, we use $\pm f$

to denote f or $\overline{f}$. Let $v \in V$ be a concrete state variable, we use $v' \in V'$ to denote the corresponding next state variable. If f is a boolean function over V, then f' is the same function over V'. In this subsection, when it is clear from the context, we sometimes omit timed variables. For example we use $R \wedge \gamma(ce_0) \rightarrow \gamma(\overline{ce}_1)$ to really mean $R^0 \wedge \gamma(ce_0) \rightarrow \gamma(\overline{ce}_1)$.

The flattened branch conditions, which have not yet been added as predicates, are called the *candidate predicates.* A naive algorithm to compute the required set of invalidating predicates is the following: First, the set of candidate predicates is ordered according to some importance criteria. Using this order, candidate predicates can be added to the abstract model one at a time and the given counterexample can be checked on the refined abstract model. If the counterexample is invalidated, the already added candidate predicates will be the required set of invalidating predicates. This naive algorithm has two disadvantages. One is that the order of the predicates affects the size of the result. A bad order may prevent the discovery of a smaller number of invalidating predicates. Most importantly, the computation time is too high, because once a predicate is added, the abstract model has to be updated as described in Section 2.3. Instead, we have developed a new *lazy refinement* algorithm, which avoids computing the full refined abstract model at each stage. Intuitively, in this algorithm, the given abstract counterexample is extended by assigning 0, 1 or x values to the abstract variables corresponding to the candidate predicates. A candidate abstract variable is given a 0 or a 1 value at time i if it can be determined from the counterexample at time $i-1$ and i; otherwise an unknown value x is given. The counterexample is invalidated if it can not be extended to the next time step. If that is the case, we perform a backward analysis from the time of failure until time 0 to identify those candidate predicates that are responsible for this failure. The predicates identified in this manner will invalidate the spurious counterexample.

Suppose there are already m predicates in the abstract model. Let $ce = \langle ce_0, ce_1, \ldots, ce_n \rangle$ be a spurious abstract counterexample. Note that, each ce_j is a conjunction of literals over the set of abstract state variables $B_1, \ldots, B_m$. Let $cp = \{cp_{m+1}, cp_{m+2}, \ldots, cp_{m+k}\}$ be the set of candidate predicates, which are temporarily represented by abstract state variables $\{B_{m+1}, B_{m+2}, \ldots, B_{m+k}\}$ (These candidate predicates have not been added to the abstract model yet). The example in Figure 5.3 illustrates how our algorithm works. Suppose there are 2 predicates, 3 candidate predicates

	time 0	time 1	time 2
B1	1	0	1
B2	0	1	1
B3	1	1	
B4	0	x	
B5	0	1	

Figure 5.3: A refinement example

and a spurious abstract counterexample of length 3. The counterexample contains values for predicates P_1 and P_2 at each time from 0 to 2. Our algorithm first determines the values for the candidate predicates at time 0. If $(S_0 \wedge \gamma(ce_0)) \rightarrow \overline{cp_4}$ is a tautology, then any valid extension of ce_0 must have the abstract variable corresponding to cp_4 set to 0. The values of other candidate predicates at time 0 can be determined similarly. The resulting extended counterexample at time 0 is denoted by ece_0. We then extend the counterexample at time 1 to obtain ece_1. For example, if we can prove that

$$(R \wedge \gamma(ce_0) \wedge cp_3 \wedge \overline{cp_4} \wedge \overline{cp_5} \wedge \gamma(ce_1)) \rightarrow cp'_3 \tag{5.6}$$

is a tautology (where cp'_3 is the same as cp_3 except that it is over the next state variables), the value of this candidate predicate must be 1. Note that we can not determine the value of cp_4 at time 1, therefore its value is unknown in the extended counterexample. After ece_1 is determined, if

$$(R \wedge \gamma(ce_1) \wedge cp_3 \wedge cp_5) \rightarrow \gamma(\overline{ce_2}) \tag{5.7}$$

is a tautology, then the counterexample can not be extended to time 2, thus it has been invalidated. Finally, we identify the set of invalidating predicates. It is possible that not all candidate predicates in the left hand side of equations (5.6) and (5.7) are necessary in showing that they are tautologies. Only those in the proof of the tautologies are necessary. Proofs can be obtained from the SAT conflict dependency analysis described in Chapter 3. Suppose, we can determine that cp_3, cp_5 in equation (5.6) and cp_5 in equation (5.7) are not in the respective proofs for those two implications. Then we can deduce that, of all candidate predicates, cp_3 alone is responsible for disabling the transition from time step 1 to time step 2 (since cp_5 is not needed in the proof of equation (5.7)). Moreover, of all candidate predicates, only cp_4 at time 0 determines the value of cp_3 at time step 1 (since cp_3, cp_5 do not appear in the proof of equation (5.6)). Thus the set of invalidating predicates is $\{cp_3$, $cp_4\}$. Note, we have worked backwards along the counterexample. We first found some invalidating predicates at time step 1 and then used that to find more invalidating predicates at time step 0. This is the basic idea of our algorithm to find the set of invalidating predicates.

We now present the lazy refinement algorithm in detail. Our algorithm is separated into three parts, the first one, which computes ece_0, is shown in Figure 5.4. The second one, which computes ece_{i+1} making use of ece_i, is shown in Figure 5.5. The last one, shown in Figure 5.6, computes the invalidating predicates as a subset of the candidate predicates once the counterexample is invalidated.

```
COMPUTE_INITIAL
1 let $ece_0 = ce_0$
2 for each candidate predicate $cp_{m+j}$
3         if $(S_0 \wedge \gamma(ce_0)) \rightarrow cp_{m+j}$ is a tautology
4                 let $ece_0 = ece_0 \wedge B_{m+j}$
5         elseif $(S_0 \wedge \gamma(ce_0)) \rightarrow \overline{cp_{m+j}}$ is a tautology
6                 let $ece_0 = ece_0 \wedge \overline{B_{m+j}}$
7         endif
8 endfor
```

Figure 5.4: Algorithm to compute ece_0

The algorithm to compute ece_0 is similar to the algorithm for computing the set of abstract initial states in Section 2.3, except that we use $S_0 \wedge \gamma(ce_0)$ instead of S_0 alone. This makes sense because our goal is to extend the current counterexample. The idea is to determine if the set of concrete initial states S_0 and the concrete states corresponding to ce_0 can imply either the truth or falsity of each candidate predicate; otherwise the value of the candidate predicate is unknown.

Given the extended counterexample at time i, the algorithm in Figure 5.5

extends the counterexample to time $i+1$. It first checks whether there are any concrete transitions between $\gamma(ece_i)$ and $\gamma(ce_{i+1})$. The code for this is given in lines (1) to (4). If it is not the case, the counterexample has been invalidated by the candidate predicates, the set of invalidating predicates is calculated and returned in line (3). If it is possible to make a concrete transition from $\gamma(ece_i)$ to $\gamma(ce_{i+1})$, the algorithm will check whether a candidate predicate is guaranteed to be true/false for such concrete transitions. This is computed in line (7) and line (9) and ece_{i+1} is updated. If the counterexample can be extended from time 0 until time n, the set of flattened branch conditions are not enough to invalidate the counterexample. We will resort to the traditional refinement methods as described in this chapter to compute a new predicate.

```
       //$i$: time to extend counterexample
       COMPUTE_NEXT($i$)
    1  **if** $(R \wedge \gamma(ece_i)) \rightarrow \gamma(\overline{ce_{i+1}})$ is a tautology
    2          let $f = (R \wedge \gamma(ece_i)) \rightarrow \gamma(\overline{ce_{i+1}})$
    3          **return** DETERMINE_PREDICATES($i$, $f$)
    4  **endif**
    5  let $ece_{i+1} = ce_{i+1}$
    6  **for** each candidate predicate $cp_{m+j}$
    7          **if** $(R \wedge \gamma(ece_i) \wedge \gamma(ce_{i+1})) \rightarrow cp'_{m+j}$ is a tautology
    8                  let $ece_{i+1} = ece_{i+1} \wedge B_{m+j}$
    9          **elseif** $(R \wedge \gamma(ece_i) \wedge \gamma(ce_{i+1})) \rightarrow \overline{cp'_{m+j}}$ is a
               tautology
   10                  let $ece_{i+1} = ece_{i+1} \wedge \overline{B_{m+j}}$
   11          **endif**
   12  **endfor**
```

Figure 5.5: Algorithm to compute ece_{i+1}

If the counterexample is invalidated at line (1) in Figure 5.5, the algo-

```
   //$t$: the time when extending counterexample fails
   //$f = (R \wedge \gamma(ece_t)) \rightarrow \gamma(\overline{ce_{t+1}})$
   DETERMINE_PREDICATES($t$, $f$)
1  let $np = \{\langle \pm B_{m+j}, t\rangle \mid \pm cp_{m+j}$ is in the proof of $f\}$
2  for $i = t - 1$ to 0
3          let $taut(i) = \{(R \wedge \gamma(ece_i) \wedge \gamma(ce_{i+1})) \rightarrow \pm cp'_{m+q} \mid$
                              $\langle \pm B_{m+q}, i+1\rangle \in np\}$
4          let $prf = \{$ proofs for the implications in $taut(i)\}$
5          let $np = np \cup \{\langle \pm B_{m+w}, i\rangle \mid$
                              $\pm cp_{m+w}$ is in any proof in $prf\}$
6  endfor
7  return $\{cp_{m+j} \mid \exists 0 \leq i \leq t.\ \langle \pm B_{m+j}, i\rangle \in np\}$
```

Figure 5.6: Algorithm to compute invalidating predicates

rithm in Figure 5.6 is called with the time t and $f = (R \wedge \gamma(ece_t)) \rightarrow \gamma(\overline{ce_{t+1}})$. We use the set *np* to hold all candidate predicates that are given a 0 or 1 value in the time steps preceding t and result in the failure of the counterexample. In line (1), *np* is initialized to all candidate predicates that are directly responsible for the failure. This is done by analyzing the proof for the failure of the counterexample. In the loop between line (2) and line (6), we go backward in time to find the set of candidate predicates that are indirectly responsible for the failure. Finally in line (7), the set of invalidating predicates is returned. Note that, in line (3), $taut(i)$ is a subset of the tautologies we computed from the algorithm in Figure 5.5. For each implication $(R \wedge \gamma(ece_i) \wedge \gamma(ce_{i+1})) \rightarrow \pm cp'_{m+q}$ in $taut(i)$, we refine the abstract transition relation $\hat{R}$ by conjuncting it with $ece_i \rightarrow (\overline{ce_{i+1}} \vee \pm B'_{m+q})$. Therefore, our algorithm not only computes the subset of the flattened branch conditions which can invalidate the given spurious abstract counterexample but also computes the refined abstract model. Our algorithm does not build the whole refined abstract model and then test whether it invalidates the counterexample. Instead, it gradually refines the abstract model until the counterexample is invalidated. Therefore, our lazy algorithm can be more efficient than the naive algorithm.

5.4 Experimental Results

We have implemented our predicate abstraction refinement framework on top of NuSMV model checker [20]. We modified the SAT checker zChaff to support conflict dependency analysis. We have developed a simple translator for Verilog. Given a RTL Verilog design, it can generate an equivalent design where the branch conditions are replaced using unique signals. This translator is based on the Icarus Verilog simulator and synthesis tool kit [74]. We have also developed synthesis scripts for Synopsys Design Compiler [41] to translate a RTL Verilog design to gate level Verilog design. Then we use the Ver structural Verilog compiler to convert gate level Verilog design to the IVF format. Finally, we have developed several perl scripts to translate a design in IVF format to a SMV file. These scripts are available at [71]. All experiments were performed on a dual 1.5GHz Athlon machine with 3GB of RAM running Linux.

We have two verification benchmarks: one is the integer unit (IU) of the picoJava microprocessor from Sun; the other is a programmable FIR filter (PFIR) which is a component of a system-on-chip design. All properties verified were simple **AG** properties. We enable dynamic BDD variable reordering and cone of influence reduction during verification. In Table 5.1, the second and third columns show the number of registers and gates in the COI of each property. We compare three abstraction refinement systems, including the BDD based aSMV [23], the SAT based localization reduction [18] (SLOCAL), and the SAT based predicate abstraction (SPRED) described in this paper. The detailed results obtained using aSMV are not listed in Table 5.1 because aSMV can not solve any of the properties within the 24hr time limit. This is not surprising because aSMV is based on BDD based image computation and it can only handle circuits with hundreds of state variables, that too provided good initial variable ordering. Since the time to generate good BDD variable orderings can be substantial, we did not pre-generate them for any of the properties. Another limitation of aSMV is that the current implementation (described in [23]) can not extract useful predicates from RTL Verilog. For the first four properties from IU, SLOCAL takes about twice the time taken by SPRED. Furthermore, the number of register in the final abstract models from SLOCAL are much larger than the number of predicates in the final abstract models from SPRED. For the rest of the four properties from PFIR, SLOCAL can not solve any of them in 24 hours because all the abstract models had around 100 registers. SPRED could solve each of them easily using about 50 predicates. A detailed analysis of the PFIR results shows that the extraction algorithm extracted about 9 branch conditions from the RTL Verilog, which were later used as predicates. Without these extracted predicates, the set of predicates computed using traditional refinement algorithm was not sufficient to finish verification within 24 hours for these four properties from PFIR.

circuit	# regs	# gates	ctrex length	Localization			Predicate Abstraction		
				time	iters	regs	time	iters	pred
IUscr2	4855	149143	20	29115.0	69	115	13515.0	22	14
IUscr3	4855	149143	true	4794.1	9	31	2003.0	10	6
IUscr7	4855	149143	12	7332.1	17	73	3869.8	10	8
IUprop4	4855	149143	8	5603.7	36	61	3495.9	13	9
PFIRprop8	244	2304	true	> 24 hours	>37	>91	288.5	68	35
PFIRprop9	244	2304	true	>24 hours	>33	>85	2448.7	146	46
PFIRprop10	244	2304	true	>24 hours	>46	>94	6229.3	161	55
PFIRprop12	247	2317	true	>24 hours	>46	>91	707.0	111	45

Table 5.1: Comparison between localization reduction and predicate abstraction.

Chapter 6

Combine Localization Reduction with Predicate Abstraction

It is usually the case that verification effort is focused more on the control logic than the data computation because most bugs exist in designing the control logic. Traditional predicate abstraction techniques can perform badly when verifying hardware systems which contain extensive control structure (*control intensive systems*). The control logic usually consists of concurrent state machines. Each of these state machines may depend on several *control variables*, that encode the change of state. Since the behavior of a control intensive system is determined to a large extent by the control variables, the number of predicates over the control variables that are needed can be much larger than the number of control variables. In such a case, it is better to use the control variables as predicates, (called *variable predicates*), instead of the original predicates (called *original or formula predicates*). We propose a clustering based heuristic to identify important control variables and retain these control variables in the abstract model. By doing this we also circumvent to a certain extent the problem of building the abstract model. This method works extremely well in practice.

It is usually the case that different predicates are not independent. We describe efficient methods to compute constraints between predicates, which are added as invariants to the abstract model to make it more accurate.

6.1 Identifying Control Variables

Predicate abstraction is suitable for handling variables with large domains. Such variables are usually called *data variables.* By replacing important formulas over concrete data variables with abstract predicates, it is possible to reduce the complexity of verification significantly. Besides data variables, there are other variables with small domains (e.g., boolean variables) that control the behavior of the system to be verified. These variables are called *control variables.* Abstracting control variables does not give much advantage. Because control variables typically have small domains, the amount of reduction obtained by replacing a predicate over several control variables with an abstract boolean variable is not very significant.

We propose a clustering-based heuristic to identify the important control variables for the verification of the given property. Let $\{P_1, \ldots, P_k\}$ be the set of predicates. Each predicate P_i is a boolean formula over a set of concrete state variables, called the *supporting variables* of P_i. We partition predicates into small clusters. Initially, each predicate is a cluster. We merge two clusters if the intersection of their supports crosses a certain threshold (the support of a cluster is the union of the supporting variables for each predicate in the cluster). We continue this process until no more clusters can be merged. Thus, the clusters we create partition the predicates into disjoint sets (but the supporting variables of different clusters may still overlap). Let c be a cluster, the set of indexes of predicates in c be $I(c)$, the supporting variables of c be $v(c)$. In general, for each variable there are several equivalent boolean variables which encode the domain of this variable. The set of boolean variables for variables in $v(c)$ is called the set of *supporting boolean variables*. For a cluster c, if the number of predicates is comparable to the number of supporting boolean variables, then this cluster is called a *control cluster* and the supporting variables of c are regarded as control variables.

6.2 Combining with Localization Reduction

It is well known that, given n boolean variables, the number of distinct propositional formulas over them is 2^{2^n}. Since control variables determine the control flow of the system under verification, in order to approximate the behavior of the concrete system, many predicates over the control variables may be necessary. Each of these propositional formulas may become a predicate during predicate abstraction. Therefore, for the verification of control intensive systems, a blowup of the abstract model is likely when using existing predicate abstraction methods. Furthermore, building the abstract model using equations (2.5) and (2.6) is time consuming. Both these problems can be avoided by using our technique of *combining the localization reduction with predicate abstraction.* Using our method, it is possible to bound the size of the abstract model by that of the concrete model. We retain some of the control variables in the abstract model (the criteria for retaining a control variable is discussed later in this section). The concrete transition relations for these control variables also serve as abstract transition relations after some minor modifications. So we can easily build abstract transition relations for all these control variables.

The modification to the concrete transition relation is as follows: for a variable $v \in v(c)$, let R_v be the concrete transition relation for v. Let $R'_v = R_v[P_j \leftarrow B_j, \text{for all } j \text{ such that } P_j \text{ is a formula predicate}]$. That is, we replace all occurrences of every formula predicate P_j in R_v by the corresponding abstract boolean variable B_j. Then, we use R'_v as the abstract transition relation for variable v. In the terminology of localization reduction, variable v is visible and unabstracted. There is one major difference between localization reduction and our method: In localization reduction, the transition relation for a visible variable is copied from the concrete model to the abstract model, whereas in our method, we replace a subformula of the concrete transition relation if that subformula corresponds to a formula predicate. Therefore, in our modified localization reduction algorithm, the transition relations for the control variables are modified so that the abstract variables corresponding to formula predicates constrain the possible next states of the control variables. This leads to a more accurate model.

Note that the abstract model built using the localization reduction has more primary inputs (invisible variables) than the abstract model built using predicate abstraction. Therefore, we retain unabstracted only those variables whose next state logic has a small number of inputs.

6.3 Correlations between Control Variables and Predicates

Our abstract model includes real predicates and control variables. In this section, a method to correlate predicates and control variables will be discussed. Recall from Section 6.1 that the clusters we build partition the predicates into disjoint sets (although the supporting variables of the clusters may overlap). Our method replaces the predicates in the control clusters by the supporting variables. There might be other predicates which have these control variables in their support. As an example, suppose we decide to drop a predicate cluster $\{P_1 \equiv x \vee y, P_2 \equiv x \wedge y\}$ and replace the two predicates with the variables $\{x, y\}$. Suppose also there are two additional predicates, $P_3 \equiv x \vee y \vee z$ and $P_4 \equiv x \wedge y \wedge w$ whose corresponding abstract state variables are B_3 and B_4, respectively. Thus, the abstract state variables include x, y, B_3, B_4. Further assume that the next state value for variable x is defined as $\neg z$ in the concrete model. Note that the values of variables x, y and values of B_3, B_4 are not independent. The following are three possible scenarios:

- If we know that B_3 is false in an abstract state, then $x =$ false and $y =$ false in that state.
- If we know $x =$ false in an abstract state, then B_4 must be false in that state.

- If we know B_3 is false in an abstract state, then in the corresponding concrete states, z is false. Therefore, in the abstract successor states, x will be true.

It is desirable to incorporate the correlations/constraints between control variables and real predicates into the abstract model. This will make the abstraction more accurate. Our method does not directly compute these constraints. Instead, we selectively introduce the concrete definitions of some predicates into the abstract model as invariants. The model checking procedure will enforce any implied constraints through these invariants. For the above example, we add z, w as two additional abstract input variables and add the definitions of the two predicates as abstract invariants: $B_3 = (x \vee y \vee z)$, and $B_4 = (x \wedge y \wedge w)$. This will force the abstract model to observe any constraints between variables x, y and B_3, B_4. Note that by doing this we have added two new variables z, w to the abstract model. This could make the abstract model larger. To overcome this problem, we add the definition of a predicate to the abstract model only if most of the variables in the support of this predicate are either control variables themselves (e.g. x, y for B_3) or in the support of control variables (e.g. z for x). In this way, the added invariants will restrict the possible values of the control variables and predicates. This will ensure we only add a small number of additional variables, e.g., z and w.

6.4 Correlations Between Formula Predicates

It is also possible that the predicates in a non-control cluster may not be independent, in the sense that not all possible combinations of assignments to their abstract state variables are possible. For the example in the previous section, when $B_3 =$ false, B_4 must also be false. For a given cluster c, let $v(c)$ be the concrete supporting variables in c, let $I(c)$ be the indexes of the predicates in c. We define $g(c)$, called the *consistent abstract states over cluster* c, as follows

$$g(c) = \{\hat{s} \mid \exists s \in S. \bigwedge_{j \in I(c)} (P_j(s) = B_j(\hat{s}))\} \tag{6.1}$$

It is easy to see that any $\hat{s} \notin g(c)$ does not have any corresponding concrete state and therefore it should be excluded from the abstract model checking. We represent the computed consistent abstract states for each non-control cluster as invariant in the abstract model. It is possible to compute a single set of consistent abstract states by conjuncting all predicates instead of conjuncting predicates of each cluster separately. Although this will result in a more accurate constraint, it may be computationally expensive when the number of predicates is large. We now show how to compute $g(c)$. Our algorithm is based on BDDs. We first build BDDs for each P_j and B_j, then $g(c)$ can be calculated by conjuncting $P_j(s) = B_j(\hat{s}), j \in I(c)$ and quantifying $v(c)$. This is not expensive because the number of predicates in a cluster is usually small.

6.5 Experimental Results

We have implemented our abstraction refinement framework on top of NuSMV model checker and the zChaff SAT solver [76].

We present the results in Table 6.1. We used two benchmarks: one is the integer unit (IU) of the picoJava microprocessor from Sun; the other is a programmable FIR filter (PFIR) which is a component of a system-on-chip design. For all the properties shown in the first column of Table 6.1, we have performed cone-of-influence reduction before the verification. The resulting number of registers and gates is shown in the second and third columns. Most properties are true, except for PFIRscr1 and PFIRprop5. The lengths of the counterexamples are shown in the fourth column. All these properties are difficult to verify for the state-of-art BDD-based model checker, Cadence SMV. Except for the two false properties, Cadence SMV can not verify any of them in 24 hours. The verification time for PFIRscr1 is 834 seconds, and for PFIRprop5 is 8418 seconds, which are worse than our results.

We compare two algorithms: one is the pure predicate abstraction (PRED), the other is the combined algorithm described in this chapter (COMB). In Table 6.1, the fifth to seventh columns are the results obtained using PRED; while the last four columns are the results obtained using COMB. We compare the time (in seconds), the number of refinement iterations, and the number of predicates in the final abstraction. In all cases, the combined algorithm outperforms pure predicate abstraction in the amount of time used, sometimes over an order of magnitude improvement is achieved. With the new method, the number of refinement iterations is usually smaller. This is because our new method can build a more accurate abstract transition relation. Building the abstract transition relation in predicate abstraction is time consuming because of the potentially exponential number of calls to a theorem prover (or a SAT solver in our case). In practice, the abstract transition relation is approximated initially and is refined as necessary during refinement. Each refinement of the abstract transition relation is an iteration of the algorithm. The combined algorithm described in this chapter replaces predicates with the supporting concrete state variables. The abstract transition relation for these variables is directly copied from the concrete model. Thus, these concrete state variables are not abstracted at all. Since there is no need to refine the transition relation for these variables, the number of refinement iterations can be much smaller. The number of predicates in the final abstraction is usually smaller using COMB than PRED. Two exceptions are the first two properties in Table 6.1. The reason is that we have replaced some predicates by the concrete state variables for those two properties. As described in Section 6.2, for a cluster, when the number of predicates is bigger than the number of supporting variables, the predicates are replaced by the supporting variables. In practice, we perform this replacement even if the number of predicates are slightly smaller. By doing this we can avoid generating any new predicates with the same supporting variables, thus reduce the run time. But it is possible that no such predicates are generated later during verification. As a result, this early replacement actually slightly increases the number of predicates.

circuit	# regs	# gates	ctrex length	PRED			COMB		
				time	iters	pred	time	iters	pred
IUscr1	4855	149143	true	2000.6	11	7	1218.6	7	18
IUscr3	4855	149143	true	2003.0	10	6	1466.6	10	15
IUscr6	4855	149143	true	9976.1	27	12	3498.2	20	11
PFIRscr1	243	2295	16	637.5	103	40	386.4	67	34
PFIRprop5	250	2342	17	2262.0	131	48	756.2	101	44
PFIRprop8	244	2304	true	288.5	68	35	159.8	40	25
PFIRprop9	244	2304	true	2448.7	146	46	202.7	43	27
PFIRprop10	244	2304	true	6229.3	161	55	178.2	50	25
PFIRprop12	247	2317	true	707.0	111	45	591.2	80	38

Table 6.1: Compare the pure predicate abstraction (PRED) with the combined algorithm (COMB)

Chapter 7

Removing Redundant Predicates

In predicate abstraction, the number of predicates affects the overall performance. Since each predicate corresponds to a boolean state variable in the abstract model, the number of predicates directly determines the complexity of building and checking the abstract model. Most predicate abstraction systems build an abstract model of the system to be verified. While building the abstract model, the number of calls made to a theorem prover (or a SAT solver in our case) can be exponential in the number of predicates. Consequently, it is desirable to use as few predicates as possible. Existing techniques for choosing relevant predicates may use more predicates than necessary to verify a given property. That is some of the predicates used can be redundant (the precise definition of redundancy is given later).

Counterexample guided abstraction refinement (CEGAR) [23, 44, 61] is an example of a commonly used abstraction technique. It works by introducing new predicates to eliminate spurious counterexamples. The new predicates depend on certain abstract states in the spurious abstract counterexample. Thus, different predicates are likely to be closely related when similar abstract counterexamples occur and this might lead to redundancy in the predicate set. These similarities may result in the following two cases: (a) A predicate may be logically equivalent to a propositional formula in terms of other predicates. (b) For the predicate P under consideration, there exist two nontrivial propositional formulas P_{sub} and P_{sup} in terms of other predicates such that P_{sub} implies P and P implies P_{sup}. It is obvious that when case (a) happens, the predicate is redundant. This predicate can be replaced by the equivalent formula and we thus obtain a new abstract model. We call the original abstract model the *current/original abstract model* and the new one the *reduced abstract model*. It is easy to show that the two models are bisimilar. In the other case, a predicate P satisfying case (b) may not be redundant. More conditions on the abstract model are needed to ensure that replacing P by P_{sub} or P_{sup} will not affect the results of model checking the abstract model.   We have identified two redundancy conditions for case (b), one that preserves safety properties (that is the original and the reduced abstract models both satisfy the same safety properties) and one that preserves bisimulation equivalence (that is the original and the reduced abstract models are bisimulation equivalent).   different situations and there are cases where one works better than the other. Altogether there are three different redundancy conditions. One useful feature of our redundancy conditions is that they do not require exact computation, we can use approximations and still identify redundancy.

Removing a predicate involves constructing the abstract model using the reduced predicate set. We give a simple method to construct the reduced abstract model from the original abstract model in Section 7.2.

Related Work. The notion of redundancy has been explored in resolution theorem proving [17], where it is called *subsumption*. Intuitively a clause is considered redundant if it is logically implied via substitution by other clauses. Our conditions for redundancy are more complicated. Even if a predicate is implied by other predicates, we still need to consider the abstract transition relation in order to decide whether removing the predicate will affect the results of verifying a given property.

The work that is closest to ours is the notion of *strengthening* in [6]. To build the abstract model, the weakest precondition is converted to an expression over the set of predicates in the abstraction. Thus strengthening is somewhat similar to the *replacement function* in this paper. However, in [6], the result of the strengthening is over all the predicates, while the replacement function used here is defined over a subset of the predicates. Finally, the two transformations have different purposes. Strengthening is only used to build an abstract model; while our transformation is used to remove redundant predicates and thus reduce the complexity of the abstract model.

7.1 The Replacement Function

Our goal is to eliminate B_i from the abstract model $\hat{M}$ without sacrificing accuracy. For this purpose, we define an under-approximation, $\mathcal{F}_U(B_i)$, for B_i in terms of the other variables. More precisely, let M be a concrete transition system, $\{P_1, P_2, .., P_k\}$ be a set of predicates defined on the states of M, and let ρ be a total function defined by equation (2.4). Also, let $\hat{M}$ be the corresponding abstract transition system over $V = \{B_1, B_2, .., B_k\}$. The support of an abstract set of states $\hat{S}_1$ includes B_i if and only if

$$\exists \hat{s} \in \hat{S}_1 . \hat{s}[B_i \leftarrow 0] \in \hat{S}_1 \not\Leftrightarrow \hat{s}[B_i \leftarrow 1] \in \hat{S}_1.$$

where $\hat{s}[B_i \leftarrow 0]$ is a state that agrees with $\hat{s}$ on all bits except possibly the bit B_i, which is fixed to 0. $\hat{s}[B_i \leftarrow 1]$ is similar. Consider the boolean variable B_i and the set $U = V \setminus \{B_i\}$. Let Φ denote either B_i or $\neg B_i$. The *replacement function* for Φ, denoted by $\mathcal{F}_U(\Phi)$, is defined as the largest set of *consistent* abstract states (we call an abstract state *consistent* if its concretization is not empty) whose support is included in U and whose concretization is a subset of $\gamma(\Phi)$. The implications $\gamma(\mathcal{F}_U(B_i)) \rightarrow \gamma(B_i)$ and $\gamma(\mathcal{F}_U(\neg B_i)) \rightarrow \gamma(\neg B_i)$ follow from this definition. Figure 7.1 shows the relationship between the concretization of a predicate B_i, $\mathcal{F}_U(B_i)$, and $\neg\mathcal{F}_U(\neg B_i)$.

We now show how to compute $\mathcal{F}_U(B_i)$. Consider the abstract state space $\hat{S}$ given by tuples of the form $(B_1, B_2, .., B_k)$. Not all the abstract states have corresponding concrete states. We consider only the set of consistent abstract states, g, that are related to some concrete states by the relation (2.4) in Section 2.3. Formally, if S is the set of concrete states, $\{P_j | 1 \leq j \leq k\}$

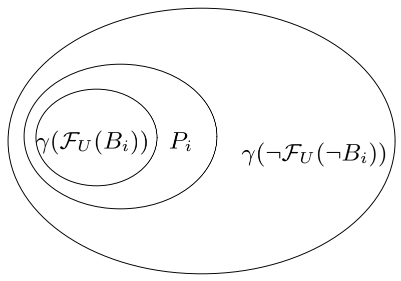
$\gamma(\mathcal{F}_U(B_i))$
P_i
$\gamma(\neg\mathcal{F}_U(\neg B_i))$

Figure 7.1: Relationship between the concretization of B_i, $\mathcal{F}_U(B_i)$, and $\neg\mathcal{F}_U(\neg B_i)$

is the set of predicates and ρ is the simulation relation as in Section 2.3, then

$$g = post[\rho](\text{true}) = \{\hat{s} \mid \exists s \in S. \bigwedge_{1\leq j\leq k} P_j(s) \Leftrightarrow B_j(\hat{s})\}.$$

For hardware verification, all the concrete state variables have finite domain. The set g can be efficiently computed using OBDDs [14] through a series of conjunction and quantification operations. We define $g\,|_{B_i}$ to be the set of reduced abstract states obtained by taking all the states in g that have the bit B_i equal to 1 and dropping the bit B_i. Similarly $g\,|_{\neg B_i}$ is obtained by taking all those states in g with bit B_i equal to 0 and dropping the bit B_i. The following theorem shows that the set $(g|_{B_i} \wedge \neg g|_{\neg B_i})$ is a candidate for $\mathcal{F}_U(B_i)$.

Theorem 7.1.1 Let $V = \{B_1, B_2, \ldots, B_k\}$ be the boolean variables. Let $U = V \setminus \{B_i\}$, and let $f_1 = g\,|_{B_i} \wedge\neg g\,|_{\neg B_i}$ be a set of abstract states. Then $\gamma(f_1) \Rightarrow \gamma(B_i)$ and f_1 is the largest set of consistent abstract states that does not have bit B_i in its support. Likewise, if $f_2 = g\,|_{\neg B_i} \wedge\neg g\,|_{B_i}$, then $\gamma(f_2) \Rightarrow \gamma(\neg B_i)$ and f_2 is the largest set of consistent abstract states that does not have bit B_i in its support.

Proof: We first prove that $\gamma(f_1) \Rightarrow \gamma(B_i)$. According to the definition of $g|_{\neg B_i}$

$$g|_{\neg B_i} = \{s_r \mid \exists s \in S. \neg P_i(s) \wedge (\bigwedge_{1 \leq j \leq k, j \neq i} P_j(s) \Leftrightarrow B_j(s_r))\}.$$

Therefore, $\neg g|_{\neg B_i} = \{s_r \mid \forall s \in S.(s \in \gamma(s_r)) \rightarrow (s \in \gamma(B_i))\}$. So $\gamma(\neg g|_{\neg B_i}) \Rightarrow \gamma(B_i)$. Since $f_1 \subseteq \neg g|_{\neg B_i}$, we have $\gamma(f_1) \Rightarrow \gamma(B_i)$.

Next, we prove that f_1 is the largest consistent set of abstract states on U such that $\gamma(f_1) \Rightarrow \gamma(B_i)$. It is easy to show that

$$f_1 = (g|_{B_i} \vee g|_{\neg B_i}) \wedge \neg g|_{\neg B_i} = (\exists B_i \; g) \wedge \neg g|_{\neg B_i} .$$

We will prove the result by contradiction. Assume there is a consistent abstract state $s_r \notin f_1$ such that $\gamma(s_r) \Rightarrow \gamma(B_i)$. According to the definition of $\neg g|_{\neg B_i}$, $s_r \in \neg g|_{\neg B_i}$. It is easy to see that $(\exists B_i \; g)$ is the largest consistent set of abstract states on U, so $s_r \in \exists B_i \; g$. This contradicts with the assumption that $s_r \notin f_1$. Therefore, f_1 is the required set. ∎

Replacement function is used extensively in the later sections. The correctness of our algorithms only depend on the property that $\gamma(\mathcal{F}_U(B_i)) \rightarrow \gamma(B_i)$. The nice advantage of this is we can use any f that satisfies $\gamma(f) \rightarrow \gamma(B_i)$ instead of using $\mathcal{F}_U(B_i) = f_1$, which is difficult to compute when there are many predicates. Instead we use the following approximation: we first partition predicates into clusters as in Section 6.1, then compute the set of consistent abstract states and replacement function for each cluster separately. We use these easy to compute approximations to identify and remove redundant predicates. This does not affect the correctness (i.e., every identified predicate is indeed redundant), but some redundant predicates may fail to be identified.

7.2 Removing Redundant Predicates

The removal of a predicate involves constructing a new abstract transition system from the old abstract transition system. The state space of the new abstract transition system is the set of all possible valuations of the boolean variables corresponding to the new predicate set. The new predicate set has one less predicate than the old predicate set. Let P_i be the redundant predicate that is to be removed. If the old state space is given by k-tuples $(B_1, B_2, ..., B_k)$, then the new state space is given by *(k-1)*-tuples $(B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_k)$. Suppose the original abstract model is $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$. We now describe how to construct the new abstract model, $M_r = (S_r, S_{0r}, R_r, L_r)$ (r for "reduced"), from $\hat{M}$ if we decide to drop the predicate P_i. The relation ρ_r between the concrete state space and the reduced state space is

$$\rho_r(s, s_r) = \bigwedge_{1 \leq j \leq k \wedge j \neq i} P_j(s) \Leftrightarrow B_j(s_r).$$

The construction of the new state space is straightforward: we just drop the boolean variable B_i. The labeling L_r is as described in Section 2.3: a reduced abstract state s_r is labeled with a predicate P_j if and only if the corresponding bit B_j is 1 in that state. The new transition relation R_r is obtained from the original abstract transition relation $\hat{R}$ by the following equation

$$R_r(s_r,\ s'_r) = \exists b_i, b'_i.\ \hat{R}(\langle s_r, b_i\rangle, \langle s'_r, b'_i\rangle) \tag{7.1}$$

where $\langle s_r, b_i\rangle$ stands for the state (in the original abstract model) obtained by inserting b_i into s_r as the i-th bit. Thus two reduced abstract states are related if there are two related states in the original abstract model that are "extensions" of these reduced abstract states. The reduced initial set of states can be similarly constructed using existential quantification as follows

$$S_{0r}(s_r) = \exists b_i.\ \hat{S}_0(\langle s_r, b_i\rangle) \tag{7.2}$$

Lemma 7.2.1 *The transition relation of the reduced abstract model defined by equation (7.1) is the same as the one built directly from the concrete model using equation (2.2) and ρ_r over the reduced set of predicates.*

Proof: Let the transition relation constructed directly be

$$\exists s\ s'.\ \rho_r(s, s_r) \wedge \rho_r(s', s'_r) \wedge R(s, s').$$

Here ρ_r is the relation between the concrete state space and the new abstract state space. Now, following equation (7.1)

$R_r(s_r,\ s'_r)$

$\equiv\ \exists b_i, b'_i.\ \hat{R}(\langle s_r, b_i\rangle, \langle s'_r, b'_i\rangle)$

$\equiv\quad \exists b_i, b'_i. \exists s\ s'. \rho(s, \langle s_r, b_i\rangle) \wedge \rho(s', \langle s'_r, b'_i\rangle) \wedge R(s, s')$

$\equiv\quad \exists b_i, b'_i. \exists s,\ s'. (P_i(s) \Leftrightarrow b_i) \wedge (P'_i(s') \Leftrightarrow b'_i) \wedge \rho_r(s, s_r) \wedge \rho_r(s', s'_r) \wedge R(s, s')$

(since $\rho(s, \hat{s}) \equiv \bigwedge_{1\le i\le k} P_i(s) \Leftrightarrow B_i(\hat{s})$)

$\equiv\quad \exists s, s'. (\exists b_i, b'_i. (P_i(s) \Leftrightarrow b_i) \wedge (P'_i(s') \Leftrightarrow b'_i)) \wedge \rho_r(s, s_r) \wedge \rho_r(s', s'_r) \wedge R(s, s')$

$\equiv\quad \exists s, s'. \rho_r(s, s_r) \wedge \rho_r(s', s'_r) \wedge R(s, s')$

(since $\exists b_i, b'_i. (P_i(s) \Leftrightarrow b_i) \wedge (P'_i(s') \Leftrightarrow b'_i)$ is a tautology).

The last expression is equivalent to the transition relation constructed directly using (2.2). ∎

Thus, R_r constructed using equation (7.1) is equivalent to the one constructed directly from the concrete model using equation (2.2).

7.3 Redundant Predicates for Safety Properties

A predicate in a given set of predicates is *redundant* for a set of properties in $\mathcal{L}$ if the abstract transition system constructed without using this predicate satisfies the same set of properties as the original abstract transition system (constructed using all the predicates). In this section we deal with safety properties of the form $\mathbf{AG}\, p$, where p is a boolean formula without temporal operators. Note that any safety property can be rewritten into the above form through tableau construction with no fairness constraints [24].

Let $\hat{S}$ be a set of states defined by a set of boolean variables $V = \{B_1, B_2, .., B_k\}$ as before, and $U = V \setminus \{B_i\}$. Let $S_r = \mathit{proj}[U](\hat{S})$ denote the projection of the set $\hat{S}$ on U. For any state $s_r \in S_r$, $\mathit{extend}[B_i](s_r)$ is a set of states defined as follows:

- If $\mathcal{F}_U(B_i)(s_r)$, then $\mathit{extend}[B_i](s_r) \;=\; \{\langle s_r, 1\rangle\}$.
- If $\mathcal{F}_U(\neg B_i)(s_r)$, then $\mathit{extend}[B_i](s_r) \;=\; \{\langle s_r, 0\rangle\}$.
- If $\neg\mathcal{F}_U(B_i)(s_r) \wedge \neg\mathcal{F}_U(\neg B_i)(s_r)$, $\mathit{extend}[B_i](s_r) \;=\; \{\langle s_r, 0\rangle,\ \langle s_r, 1\rangle\}$.

We say that a set of consistent abstract states $\hat{S}$ is *oblivious* to B_i if and only if

$$\forall \hat{s} \in \hat{S}.\ (\neg\mathcal{F}_U(B_i)(\hat{s}) \wedge \neg\mathcal{F}_U(\neg B_i)(\hat{s})) \Rightarrow (\hat{s}[B_i \leftarrow 0] \in \hat{S} \wedge \hat{s}[B_i \leftarrow 1] \in \hat{S})$$

Intuitively, if neither $\mathcal{F}_U(B_i)(\hat{s})$ nor $\mathcal{F}_U(\neg B_i)(\hat{s}))$ holds, the values of variables $B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_k$ can not determine the value of B_i. In order for $\hat{S}$ to be oblivious, it must contain states with both possible values of B_i.

Lemma 7.3.1 *Given a set of states $\hat{S_1} \subseteq \hat{S}$, such that $\hat{S_1}$ is oblivious to a predicate B_i, then $\mathit{extend}[B_i](\mathit{proj}[U](\hat{S_1})) \equiv \hat{S_1}$.*

Proof: The proof follows from the definitions of $\mathit{extend}[B_i](s_r)$ and oblivious set.

- First we prove $extend[B_i](proj[U](\hat{S}_1)) \subseteq \hat{S}_1$. Let $\langle s_r, b_i\rangle$ be a state in $extend[B_i](proj[U](\hat{S}_1))$. Therefore, $s_r \in proj[U](\hat{S}_1)$. Let $\langle s_r, b_i'\rangle$ be the state in $\hat{S}_1$ whose projection is s_r.
  - If $\mathcal{F}_U(B_i)(s_r)$, according to the definition of extend, b_i must be 1. Since $\langle s_r, b_i'\rangle$ is a consistent abstract state, b_i' must be 1. Therefore $\langle s_r, b_i\rangle \in \hat{S}_1$.
  - If $\mathcal{F}_U(\neg B_i)(s_r)$, the proof is similar to the previous case.
  - Otherwise, since $\hat{S}_1$ is oblivious to B_i, the two states $\langle s_r, 0\rangle$ and $\langle s_r, 1\rangle$ are all in $\hat{S}_1$. Therefore $\langle s_r, b_i\rangle \in \hat{S}_1$.
- Now we prove $\hat{S}_1 \subseteq extend[B_i](proj[U](\hat{S}_1))$. Let $\langle s_r, b_i\rangle$ be a consistent state in $\hat{S}_1$. So $s_r \in proj[U](\hat{S}_1)$. Based on the definition of extend, it is easy to see $\langle s_r, b_i\rangle \in extend[B_i](proj[U](\hat{S}_1))$.

Hence, $extend[B_i](proj[U](\hat{S}_1)) \equiv \hat{S}_1$. ▮

Lemma 7.3.2 *Given an abstract transition system $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$ which corresponds to a set of boolean variables $V = \{B_1, \ldots, B_k\}$. Let B_i be one of the variables in V and $U = V \setminus \{B_i\}$. Let $M_r = (S_r, S_{0r}, R_r, L_r)$ be the abstract transition system corresponding to U. If $\hat{S}_1$ is a set of states on V that is oblivious to B_i, then $proj[U](post[\hat{R}](\hat{S}_1))$ is the same as $post[R_r](proj[U](\hat{S}_1))$, i.e., $proj$ and $post$ commute.*

Proof: For this proof we use

$$R_r(s_r,\ s'_r) = \exists b_i, b'_i.\ \hat{R}(\langle s_r, b_i\rangle, \langle s'_r, b'_i\rangle)$$

- First we prove that $proj[U](post[\hat{R}](\hat{S}_1)) \subseteq post[R_r](proj[U](\hat{S}_1))$. Let ${s'}_r$ be a state in $proj[U](post[\hat{R}](\hat{S}_1))$. There exist two states $\hat{s}, \hat{s}'$ such that $(\hat{s} \in \hat{S}_1) \ \wedge\ \hat{R}(\hat{s}, \hat{s}') \ \wedge\ (proj[U](\hat{s}') = {s'}_r)$. Let s_r be $proj[U](\hat{s})$. Since $\hat{R}(\hat{s}, \hat{s}')$ $R_r(s_r,\ {s'}_r)$ holds. Therefore, ${s'}_r \in post[R_r](proj[U](\hat{S}_1))$.

- Now we prove $post[R_r](proj[U](\hat{S}_1)) \subseteq proj[U](post[\hat{R}](\hat{S}_1))$. Let ${s'}_r$ be a state in $post[R_r](proj[U](\hat{S}_1))$. There exists $s_r \in proj[U](\hat{S}_1)$ such that $R_r(s_r, {s'}_r)$. According to the definition of R_r, there must be two states $\hat{s}, \hat{s}'$ such that $\hat{s} = \langle s_r, b_i\rangle \wedge \hat{s}' = \langle {s'}_r, b'_i\rangle \wedge \hat{R}(\hat{s}, \hat{s}')$ for some values of b_i, b'_i. Therefore, $\hat{s} \in extend[B_i](s_r)$. Since $s_r \in proj[U](\hat{S}_1)$, thus $\hat{s} \in extend[B_i](proj[U](\hat{S}_1))$. According to Lemma 7.3.1, $\hat{s} \in \hat{S}_1$. Therefore, ${s'}_r \in proj[U](post[\hat{R}](\hat{S}_1))$.

Hence, $proj[U](post[\hat{R}](\hat{S}_1)) = post[R_r](proj[U](\hat{S}_1))$. ∎

A transition relation $\hat{R} \subseteq \hat{S} \times \hat{S}$ is called oblivious to B_i, if for any state $\hat{s} \in \hat{S}$, $post[\hat{R}](\hat{s})$ is oblivious to B_i. More formally, $\hat{R}$ is oblivious to B_i if and only if

$$\begin{aligned}\forall \hat{s}, \hat{s}'[\ \ & \neg\mathcal{F}_U(B_i)(\hat{s}') \wedge \neg\mathcal{F}_U(\neg B_i)(\hat{s}') \Rightarrow \\ & (\hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 1]) \Leftrightarrow \hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 0]))]\end{aligned} \tag{7.3}$$

In order to test whether a transition relation $\hat{R}$ is oblivious to B_i or not, we take the negation of (7.3) and formulate it as a SAT instance by converting it into a CNF formula. If the CNF formula is satisfiable then we conclude that $\hat{R}$ is not oblivious otherwise it is. The negation of (7.3) is the following

$$\begin{aligned}\exists \hat{s}, \hat{s}'[\ & \neg\mathcal{F}_U(B_i)(\hat{s}') \wedge \neg\mathcal{F}_U(\neg B_i)(\hat{s}') \wedge \\ & (\hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 1]) \Leftrightarrow (\neg\hat{R})(\hat{s}, \hat{s}'[B_i \leftarrow 0]))]\end{aligned} \tag{7.4}$$

Theorem 7.3.1 Given an abstract transition system $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$ which corresponds to a set of predicates V, and a safety property $f = \mathbf{AG}\, p$, where p is a propositional formula without temporal operators. Also assume that predicate B_i is one of the predicates in V but not one of the predicates in f. If $\hat{S}_0$ and $\hat{R}$ are oblivious to B_i, then the abstract transition system corresponding to the reduced set of predicates $U = V \setminus \{B_i\}$ satisfies f if and only if $\hat{M}$ satisfies it.

Proof: Since both $\hat{S}_0$ and $\hat{R}$ are oblivious to B_i, then the set of reachable states, S_j, after j steps is oblivious to B_i for any j. Let $M_r = (S_r, S_{0r}, R_r, L_r)$ be the abstract transition system corresponding to U. According to the definition of S_{0r} and Lemma 7.3.2, the set of reachable states S_{j_r} in M_r after j steps is a projection on U of $\hat{S}_j$. Since the validity of any propositional formula p, without temporal operators, on a set of states can be determined by looking only at predicates other than P_i, all the states in $\hat{S}_j$ satisfy p if and only if all the states in S_{j_r} satisfy it. Therefore, M_r satisfies f if and only if $\hat{M}$ satisfies f. ∎

7.4 Redundant Predicates for Bisimulation Equivalence

In the previous section, the reduced abstract model M_r was such that it satisfies a safety property, if and only if $\hat{M}$ satisfies it. We can strengthen this result so that M_r is bisimulation equivalent to $\hat{M}$ by imposing slightly different conditions on $\hat{R}$.

Let $\beta \subseteq \hat{S} \times S_r$ be a relation defined such that two states $\hat{s} \in \hat{S}$ and $s_r \in S_r$ are related under β if and only if $\hat{s} \in \mathit{extend}[B_i](s_r)$, where $\mathit{extend}[B_i](s_r)$ is as defined previously. We intend to make β a bisimulation relation between $\hat{M}$ and M_r. From the construction of M_r, it is easy to see that $\hat{M} \preceq_\beta M_r$. In order for $\hat{M}$ to simulate M_r, we must make sure that for any $b_i \in \{0, 1\}$, if $\langle s_r, b_i \rangle$ is a consistent abstract state, then $\langle s_r, b_i \rangle$ can simulate s_r. If only one of $\langle s_r, 0 \rangle$ and $\langle s_r, 1 \rangle$ is a consistent state, from (7.1), it is easy to see that any successor state of s_r corresponds to a successor of the single consistent state. In order to handle the case when both $\langle s_r, 0 \rangle$ and $\langle s_r, 1 \rangle$ are consistent, we have the following condition on $\hat{R}$: for any state $\hat{s} \in \hat{S}$

$$\neg \mathcal{F}_U(B_i)(\hat{s}) \wedge \neg \mathcal{F}_U(\neg B_i)(\hat{s}) \Rightarrow \forall \hat{s}'.(\hat{R}(\hat{s}[B_i \leftarrow 0], \hat{s}') \Leftrightarrow \hat{R}(\hat{s}[B_i \leftarrow 1], \hat{s}')) \tag{7.5}$$

This condition says that if the value of B_i cannot be determined by the values of the other boolean variables, i.e., both $\hat{s}[B_i \leftarrow 0]$ and $\hat{s}[B_i \leftarrow 1]$ are consistent, then $\hat{R}$ does not distinguish between different values of the bit B_i. If $\mathcal{F}_U(B_i)(\hat{s})$ is true then we know that $\hat{s}[B_i \leftarrow 0]$ is inconsistent. If $\mathcal{F}_U(\neg B_i)(\hat{s})$ is true then we know that $\hat{s}[B_i \leftarrow 1]$ is inconsistent. In case that both of these are false (which is the condition on the left hand side of (7.5)), then we require that the successors of the states $\hat{s}[B_i \leftarrow 0]$, $\hat{s}[B_i \leftarrow 1]$ be the same. Similar to Section 7.3, to test whether $\hat{R}$ satisfies condition (7.5) or not, we test the satisfiability of its negation.

$$\begin{aligned} \exists \hat{s}, \hat{s}'.\ & \neg\mathcal{F}_U(B_i)(\hat{s}) \wedge \neg\mathcal{F}_U(\neg B_i)(\hat{s}) \wedge \\ & (\hat{R}(\hat{s}[B_i \leftarrow 0], \hat{s}') \Leftrightarrow (\neg\hat{R})(\hat{s}[B_i \leftarrow 1], \hat{s}')) \end{aligned} \tag{7.6}$$

Theorem 7.4.1 If condition (7.5) holds, then β is a bisimulation relation between $\hat{M}$ and M_r

Proof: We just need to show that $\hat{M} \preceq_\beta M_r$ and $M_r \preceq_{\beta^{-1}} \hat{M}$. It is easy to see that for any state $\hat{s} \in \mathit{extend}[B_i](s_r)$, $L(\hat{s}) \cap \{P_1, \ldots, P_{i-1}, P_{i+1}, P_k\} = L_r(s_r)$.

- We first prove $\hat{M} \preceq_\beta M_r$. Based on equations (7.1) and (7.2), M_r is an existential abstraction over $\hat{M}$ induced by the simulation relation β.

- Next we prove $M_r \preceq_{\beta^{-1}} \hat{M}$. Based on equation (5′), it is easy to see that for every state $s_r \in S_{0r}$, at least one of $\langle s_r, 0\rangle$ or $\langle s_r, 1\rangle$ must be in $\hat{S}_0$.

  We now prove that if $\beta^{-1}(s_r, \hat{s})$ and $R_r(s_r, s'_r)$ then there exists $\hat{s_1}$ such that $\hat{R}(\hat{s}, \hat{s_1})$ and $\beta^{-1}(s'_r, \hat{s_1})$. If $\beta^{-1}(s_r, \hat{s})$ and $R_r(s_r, s'_r)$, by definition of R_r, there exit b_i^0, b_i^1 such that $\hat{R}(\langle s_r, b_i^0\rangle, \langle s'_r, b_i^1\rangle)$. In case $\mathcal{F}_U(B_i)(s_r)$ is true then b_i^0 has to be 1 and $\hat{s} = \langle s_r, 1\rangle$. In case $\mathcal{F}_U(\neg b_i)(s_r)$ is true then b_i^0 has to be 0 and $\hat{s} = \langle s_r, 0\rangle$. In either case there exists $\hat{s_1} = \langle {s'}_r, b_i^1\rangle$ such that $\hat{R}(\hat{s}, \hat{s_1})$ and by definition of β, ${s'}_r$ is related to $\hat{s_1}$ under β. In case neither $\mathcal{F}_U(B_i)(s_r)$ nor $\mathcal{F}_U(\neg B_i)(s_r)$ holds, we know by condition (7.5) on $\hat{R}$ that successors of $\langle s_r, 0\rangle$ under $\hat{R}$ must be exactly the same as those $\langle s_r, 1\rangle$. $\hat{s}$ can be either $\langle s_r, 0\rangle$ or $\langle s_r, 1\rangle$. So there exists $\hat{s_1} = \langle {s'}_r, b_i^1\rangle$ such that $\hat{R}(\hat{s}, \hat{s_1})$ and by definition of β, ${s'}_r$ is related to $\hat{s_1}$ under β.

∎

It is interesting to note that the conditions for preserving safety properties and bisimulation equivalence are different and do not subsume each other. This is illustrated in the next section.

7.5 Difference in the Bisimulation and **AG** p conditions

We have seen two redundancy conditions, one for preserving **AG** p properties and the other for preserving CTL^* properties. In this section, we give examples of transition relation which satisfy one of the conditions and violates the other. This demonstrates that the conditions (7.3) and (7.5) are not comparable.

7.5.1 A transition relation that satisfies the Bisimulation condition

We first present an abstract transition relation that satisfies the Bisimulation condition, (7.5), but does not satisfy the obliviousness condition required for preserving $\mathbf{AG}\,p$ properties. The abstract transition system is:

$$\begin{aligned} (a) \quad & B_2 \rightarrow B'_1 \wedge B'_4 \\ (b) \quad & B_3 \rightarrow B'_1 \wedge B'_2 \\ (c) \quad & B_4 \rightarrow B'_4 \end{aligned}$$

Suppose we are trying to remove B_2. Assume that $\mathcal{F}_U(B_2) = \neg B_3$ and $\mathcal{F}_U(\neg B_2) = \neg B_4$. The condition for bisimulation, (7.4), then is

$$\begin{aligned} B_3 \wedge B_4 \quad \Rightarrow \quad & [((B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow \\ & ((B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4))] \end{aligned}$$

If $B_3 \wedge B_4$ is false then the condition is true. If $B_3 \wedge B_4$ is true then we need to check the validity of

$$((B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow ((B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4))\,.$$

Now from *(b)* B'_1 is true if B_3 is true and from *(c)* B'_4 is true if B_4 is true. So the problem now reduces to validity of

$$((B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow ((B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4))$$

which is trivially true. So $\hat{R}$ satisfies the bisimulation condition. Now we show that it does not satisfy the condition for $\mathbf{AG}\,p$ preservation. Condition for $\mathbf{AG}\,p$ preservation in this case would be

$$\begin{aligned} B'_3 \wedge B'_4 \;\;\Rightarrow\;\; & [((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1 \wedge 0)\;\; \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow \\ & ((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1 \wedge 1)\;\; \wedge (B_4 \rightarrow B'_4))] \end{aligned}$$

which is equivalent to

$$\begin{aligned} B'_3 \wedge B'_4 \;\;\Rightarrow\;\; & [((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow false)\;\; \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow \\ & ((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1)\;\; \wedge (B_4 \rightarrow B'_4))] \end{aligned}$$

This expression is not true for $B'_3 \;\; = B'_4 \;\; = B'_1 \;\; = B_3 \;\; = 1$ and $B_2 \;\; = B_4 \;\; = 0$. So we have shown a transition relation $\hat{R}$ that satisfies the bisimulation condition but not the $\mathbf{AG}\,p$ preservation condition.

7.5.2 A transition relation that satisfies the $\mathbf{AG}\,p$ condition

The transition relation is

$$\begin{aligned} & B_3 \rightarrow \neg B'_4 \\ & B_2 \rightarrow B'_1 \wedge B'_5 \\ & B_3 \rightarrow B'_1 \wedge \neg B'_2 \end{aligned}$$

We assume that $\mathcal{F}_U(B_2) = \neg B_3$ and $\mathcal{F}_U(\neg B_2) = \neg B_4$. The $\mathbf{AG}\,p$ preservation condition (after some simplification) is

$$\begin{aligned} {B'}_3 \wedge {B'}_4 \;\Rightarrow\; & [((B_3 \rightarrow \neg {B'}_4) \wedge (B_2 \rightarrow {B'}_1 \wedge {B'}_5) \wedge (\neg B_3)) \Leftrightarrow \\ & ((B_3 \rightarrow \neg {B'}_4) \wedge (B_2 \rightarrow {B'}_1 \wedge {B'}_5) \wedge (B_3 \rightarrow B'_1))] \end{aligned}$$

If ${B'}_3 \wedge {B'}_4$ is false then the above expression is true. In ${B'}_3 \wedge {B'}_4$ is true, then we can prove the following:

- $((B_3 \rightarrow \neg {B'}_4) \wedge (B_2 \rightarrow {B'}_1 \wedge {B'}_5) \wedge (\neg B_3))$
  $\Rightarrow ((B_3 \rightarrow \neg {B'}_4) \wedge (B_2 \rightarrow {B'}_1 \wedge {B'}_5) \wedge (B_3 \rightarrow B'_1))$. We only need to prove $\neg B_3$ implies $(B_3 \rightarrow B'_1)$, which is trivially true.

- $((B_3 \rightarrow \neg {B'}_4) \wedge (B_2 \rightarrow {B'}_1 \wedge {B'}_5) \wedge (B_3 \rightarrow B'_1))$
  $\Rightarrow ((B_3 \rightarrow \neg {B'}_4) \wedge (B_2 \rightarrow {B'}_1 \wedge {B'}_5) \wedge (\neg B_3))$. We just need to show that if ${B'}_3 \wedge {B'}_4$ is true and $(B_3 \rightarrow \neg {B'}_4)$ is true then $\neg B_3$. This is clear since ${B'}_4$ is true implies $\neg {B'}_4$ is not true. And $(B_3 \rightarrow \; false)$ can be true only if $\neg B_3$ is true.

Hence the **AG** p preservation condition is satisfied. The bisimulation condition for this example (after some simplification) is:

$$\begin{aligned} B_3 \wedge B_4 \;\Rightarrow\; & [((B_3 \rightarrow \neg {B'}_4) \wedge (B_3 \rightarrow {B'}_1 \wedge \neg {B'}_2)) \Leftrightarrow \\ & ((B_3 \rightarrow \neg {B'}_4) \wedge ({B'}_1 \wedge {B'}_5) \wedge (B_3 \rightarrow {B'}_1 \wedge \neg {B'}_2))] \end{aligned}$$

This expression is not true for $B_3 = B_4 = {B'}_1 = 1$ and ${B'}_5 = {B'}_4 = {B'}_2 = 0$.

Hence we have shown two transition relations such that they satisfy only one of the two preservation conditions. From this we can conclude that the two preservation conditions are not subsumed by each other.

Chapter 8

Conclusion and Future Work

To alleviate the state explosion problem in model checking large scale hardware designs, this thesis investigates abstraction refinement algorithms including the localization reduction and predicate abstraction.

We describe four technqiues to improve abstraction refinement:

- We give a localization reduction algorithm that uses multiple verification engines, including BDDs, ATPG, SAT and 3-value logic simulation. This algorithm was developed together with researchers at Synopsys.
- We also show how to perform efficient localization reduction and predicate abstraction using CNF unsatisfiability proofs.
- Predicate abstraction can be enhanced by extracting branch conditions from a Verilog program and using them as predicates. To make the set of predicates as small as possible, we describe an algorithm to remove redundant predicates.

- Finally, we show how to combine localization reduction with predicate abstraction.

The effectiveness of our abstraction refinement algorithms has been demonstrated on various industrial hardware designs with thousands of registers.

There are several directions for extending this work. Bounded model checking based on SAT is a powerful technique to search for design errors. By comparing bounded model checking with the methodology described in this thesis, it may be possible to discover when each technqiue is most applicable.

Given a CNF unsatisfiability proof, it is desirable to have a ranking algorithm to evaluate the relative importance of the registers in the proof. When the total number of registers in the proof is very large, such a ranking algorithm is valuable for selecting a small subset of the most important registers. Heuristics based on graph ranking algorithms used by Internet search algorithms [59] may be helpful.

In general, the proof extracted using the SAT conflict dependency analysis is not minimal. Efficient algorithms to reduce the unsatisfiability proofs are necessary. As demonstrated in [76], it is possible to reduce the number of decisions during the SAT search by learning multiple conflict clauses from a single conflict graph. However, this may slow down the SAT solver. It is desirable to reduce proof size while maintaining the speed of the SAT solver. For example, it may be possible to exclude a conflict graph if the conflict clauses generated from it can also be derived from other conflict graphs already found.

We have shown two ways to compute a predicate: One is based on separating deadend and bad states, the other is by extracting branch conditions from RTL Verilog designs. Program profiling techniques have been proposed in [32] to discover likely invariants that are important for the behavior of the system under verification. It may be possible to use these invariants as predicates in predicate abstraction.

Given a small set of variables, a *separating predicate* over these variables can be computed using a projection based SAT enumeration algorithm as describe in Section 5.2 of this thesis. McMillan has recently shown how to use the concept of *interpolation* [54] for unbounded model checking based on SAT. It would be interesting to apply his algorithm for computing the separating predicate in our algorithm.

Bibliography

[1] M. Abadi and L. Lamport. Composing specications. In *ACM Trans. on Prog. Lang. and Syst.*, pages 73–132, 1993.

[2] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic Reachability Analysis Based on SAT-Solvers. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, 2000.

[3] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital System Testing and Testable Design.* IEEE Computer Society Press, 1990.

[4] R. Alur and T. A. Henzinger. Reactive modules. In *11th annual IEEE symp. Logic in Computer Science (LICS '96)*, 1996.

[5] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *C. Courcoubetis, editor, Fifth Conference on Computer Aided Verification (CAV '93)*, Berlin, 1993.

[6] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI 2001.*

[7] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstractions for model checking c programs. In *TACAS 2001*, volume 2031 of *LNCS*, pages 268–283, April 2001.

[8] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. In *MSR Technical Report, 2000-14*.

[9] Sharon Barner, Daniel Geist, and Anna Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *CAV'02*, pages 65–77, 2002.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[11] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. Verifiying safety properties of a power pc microprocessor using symbolic model checking without bdds. In *CAV'99*, pages 60–71, 1999.

[12] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, pages 372–389, 2000.

[13] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *CAV'01*, pages 454–464, 2001.

[14] Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[15] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *A. Halaas and P. B. Denyer, editors, Proceedings of the International Conference on Very Large Scale Integration*, Edinburgh, Scotland, August 1991.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[17] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied Mathematics Series. Academic Press, New York, NY, 1973.

[18] Pankaj Chauhan, Edmund M. Clarke, Samir Sapra, , James Kukula, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *FMCAD'02*, 2002.

[19] H. Cho, G. Hachtel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate fsm traversal based on circuit analysis. *IEEE TCAD*, 15(12):1451–1464, December 1996.

[20] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In *CAV'99*, pages 495–499, 1999.

[21] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *POPL*, pages 343–354, 1992.

[22] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc. of Conference on Computer-Aided Verification (CAV'02)*, LNCS, 2002.

[23] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *Twlfth Conference on Computer Aided Verification (CAV'00)*. Springer-Verlag, July 2000.

[24] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[25] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching t ime temporal logic. In *Proc. Workshop on Logic of Programs, volume 131 of Lect. Notes in Comp. Sci.*, pages 52–71, 1981.

[26] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *CAV'01*, pages 436–453, 2001.

[27] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Technical University of Eindhoven, 1996.

[28] S. Das and D. Dill. Successive approximation of abstract transition relations. In *LICS'01*, 2001.

[29] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *CAV'99*, pages 160–171, 1999.

[30] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng, and W Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187, 2001.

[31] E.Goldberg and Y.Novikov. Berkmin: a fast and robust sat-solver. In *DATE*, 2002.

[32] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[33] Marcelo Glusman, Gila Kamhi, Sela Mador-Haim, Ranan Fraer, and Moshe Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation, 2003.

[34] Michael J. C. Gordon. The semantic challenge of Verilog HDL. In *LICS'95*, pages 136–145, 1995.

[35] Shankar G. Govindaraju and David L. Dill. Counterexample-Guided choice of projections in approximate symbolic model checking. In *ICCAD*, pages 115–119, 2000.

[36] Orna Grumberg and David E. Long. Model checking and modular vercation. In *J. C. M. Baeten and J. F. Groote, editors, CONCUR 91, volume 527 of LNCS*, August 1991.

[37] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. Sat-based image computation with application in reachability analysis. In *FMCAD'00*, pages 354–371, 2000.

[38] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[39] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: methodology and case studies. In *Proceedings of the Tenth International Conference on Computer-aided Verification (CAV 1998), Lecture Notes in Computer Science 1427*, pages 440–451, 1998.

[40] Pei-Hsin Ho, Thomas R. Shiple, Kevin Harer, James H. Kukula, Robert Damiano, Valeria M. Bertacco, Jerry Taylor, and Jiang Long. Smart Simulation Using Collaborative Formal and Simulation Engines. In *Proceedings of the IEEE international Conference on Computer Aided Design (ICCAD)*, pages 120–126, 2000.

[41] Synopsys Inc. Synopsys design compiler. *http://www.synopsys.com.*

[42] J. Kim, J. Whittemore, and K. Sakallah. On solving stack-based incremental satisfiability problems. In *International Conference on Computer Design*, 2000.

[43] R. P. Kurshan. *Computer-Aided Verification*. Princeton Univ. Press, Princeton, New Jersey, 1994.

[44] Yassine Lachnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, pages 98–112, 2001.

[45] Chen-Li Lin, Yen-Pang Lin, Dong Wang, and Salman Yussof. 18-545 advanced digital design project. *http://www.ece.cmu.edu/ ee545/f98/final_fantasy/index.html*.

[46] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design: An International Journal*, 6(1):11–44, January 1995.

[47] D. E. Long. Model checking, abstraction and compositional verification. Technical Report CMU-CS-93-178, Carnegie Mellon University, Computer Science Department, 1993.

[48] Yuan Lu. *Automatic Abstraction in Model Checking*. PhD thesis, Carnegie Mellon University, ECE Department, 2000.

[49] K. McMillan. Applying sat methods in unbounded symbolic model checking. In *CAV'02*, pages 250–264, 2002.

[50] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[51] K. L. McMillan. Verication of an implementation of tomasulo's algorithm by compositional model checking. In *A. J. Hu and M. Y. Vardi, editors, Conference on Computer-aided Verication (CAV '98), number 1427 in LNCS*, pages 100–121, 1998.

[52] K. L. McMillan. Verication of infinite state systems by compositional model checking. *http://www-cad.eecs.berkeley.edu/ kenmcmil/papers/1999-01.ps.gz*, February 1999.

[53] Ken McMillan. Applying sat methods in unbounded symbolic model checking. In *Computer-Aided Verification, CAV '02*. Springer-Verlag, 2002.

[54] K.L. McMillan. Interpolation and sat-based model checking. 2003.

[55] K.L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *To appear, TACAS'03*, 2003.

[56] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *38th ACM/IEEE Design Automation Conference(DAC)*, June 2001.

[57] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV'00*, 2000.

[58] Greg Nelson. *Techniques for Program Verification.* PhD thesis, Stanford University, 1980.

[59] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[60] Abelardo Pardo and Gary D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference*, pages 457–462, 1998.

[61] Corina Pasareanu, Matthew Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted java programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, 2001.

[62] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems, NATO ASI Series*, pages 123–144, 1984.

[63] Amir Pnueli. The temporal logic of programs. In *Proc 18th IEEE Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57, 1977.

[64] R.P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430, New York, 1989. Springer-Verlag.

[65] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV'97*, volume 1254, pages 72–83. Springer Verlag, 1997.

[66] H. Saidi and N. Shankar. Abstract and model check while you prove. In *11th Conference on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 443–454, July 1999.

[67] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a sat-solver. In *Hunt and Johnson, editors, Proc. Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000.

[68] J.P. Marques Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of the IEEE international Conference on Computer Aided Design (ICCAD)*, pages 220–227, 1996.

[69] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using bdds. In *Proceedings of the IEEE international Conference on Computer Aided Design (ICCAD)*, November 1990.

[70] W. Visser, S. Park, and J. Penix. Applying predicate abstraction to model check object-oriented programs. In *3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, August 2000.

[71] Dong Wang. Perl scripts to translate verilog to smv. *Available from: http://www.icarus.com/eda/verilog.*

[72] Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines. In *DAC'01*, 2001.

[73] Poul Frederick Williams, Armin Biere, Edmund M. Clarke, and Anubhav Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification*, pages 124–138, 2000.

[74] Stephen Williams. Icarus Verilog simulation and synthesis tool. *Available from: http://www.icarus.com/eda/verilog.*

[75] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of Design, Automation and Test in Europe (DATE2003)*, March 2003.

[76] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *ICCAD'01*, 2001.