Advances in Counterexample-Guided Abstraction/Refinement

Edmund M. Clarke, Ofer Strichman, *Editors*

September 3, 2003

CMU-CS-03-180

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This report is a collection of six articles on model checking in the abstraction/refinement framework. This framework is used by various techniques for tackling the state-space explosion problem that is frequently encountered in model checking.

Keywords: Model-checking, Abstraction-refinement

The articles collected in this report are (in order of appearance):

1. *Counterexample-guided abstraction refinement.* Clarke, Grumberg, Jha, Lu, Veith[2]

2. *SAT based Abstraction-Refinement using ILP and Machine Learning Techniques.* Clarke, Gupta, Kukula, Strichman [6]

3. *Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis.* Chauhan, Clarke, Kukula, Sapra, Veith, Wang [1]

4. *SAT based Predicate Abstraction for Hardware Verification.* Clarke, Talupur, Wang [4]

5. *High Level Verification of Control Intensive Systems Using Predicate Abstraction.* Clarke, Grumberg, Talupur,Wang [3]

6. *Verification of Hybrid Systems Based on Counterexample-Guided Abstraction Refinement.* Clarke,Fehnker,Han,Krogh,Stursberg,Theobald [5]

1 Introduction

This report is a collection of six previously published articles on counterexample-guided abstraction refinement in model checking:

1. Counterexample-guided abstraction refinement[2]
2. SAT based Abstraction-Refinement using ILP and Machine Learning Techniques [6]
3. Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis[1]
4. SAT based Predicate Abstraction for Hardware Verification [4]
5. High Level Verification of Control Intensive Systems Using Predicate Abstraction [3]
6. Verification of Hybrid Systems Based on Counterexample-Guided Abstraction Refinement [5]

The collection is meant for summarizing and comparing the various approaches to this problem that were developed in our group in the last three years. A survey of previous work on this subject by other research groups appears in the articles themselves. The abstraction-refinement framework was developed by Kurshan in the early 80's under the name *Localization Reduction*. A short description of this work appears in [7].

2 The Abstraction/Refinement Framework

Given a model M and a property φ, the abstraction/refinement framework encapsulates various automatic algorithms for finding an abstracted model $\hat{M}$ with the following two properties: first, $\hat{M}$ contains enough information for checking whether $M \models \varphi$ and second, $\hat{M}$ is as small as possible, so checking whether $\hat{M} \models \varphi$ can be done efficiently. This framework is an important tool for tackling the known state-explosion problem in model checking. The framework has four steps, as follows:

1. Generate an initial abstract model $\hat{M}$.
2. Use model checking to check whether $\hat{M} \models \varphi$. If yes, return TRUE (i.e., $M \models \varphi$).
3. Check whether the counterexample can be simulated on the concrete model. If yes, return FALSE (i.e., $M \not\models \varphi$).
4. Refine $\hat{M}$, and go to step 2.

An algorithm in this framework model checks in step 2 a finite series of models $\hat{M}_0, \ldots, \hat{M}_k$ s.t. for an arbitrary ACTL formula ψ, $\forall i \in [0 \ldots k-1]$. $\hat{M}_i \models \psi \rightarrow \hat{M}_{i+1} \models \psi$, $\hat{M}_i \neq \hat{M}_{i+1}$ and $\hat{M}_k \models \varphi \rightarrow M \models \varphi$. The series of models $\hat{M}_0, \ldots, \hat{M}_k$ is derived by defining an initial *conservative abstraction* function, and repeatedly refining it (conservative abstractions, informally, are abstractions that only add transitions to the model. This type of abstraction can result in false negatives, i.e., spurious counterexamples, but not in false positives). Note that in the worst case $\hat{M}_k = M$, which implies that this process is complete.

The techniques described in the six articles differ from each other in *(i)* the way they perform each of the four steps of the framework, and *(ii)* what type of concrete models they can be applied to. Fig. 1 briefly summarizes these differences. The article [5] is the only one that assumes a hybrid automaton, which includes discrete and continuous dynamics and hence describes an infinite-state system. The other five assume that the concrete model has a finite-state description.

The table mentions the terms 'Dead-end' and 'Bad' sets of states, which are important elements in the refinement technique applied by both [2] and [6]. While these terms are formally defined in the articles themselves, we explain them here informally. When model-checking $\hat{M}_i$ over φ results in a counterexample that can not be simulated on the concrete model M, we want the refinement process to eliminate this counterexample, i.e., we want to generate a refined model $\hat{M}_{i+1}$ such that this counterexample can not be reproduced. The reason that the counterexample can not be simulated on M is that it represents a trace that includes two consecutive abstract states s_f, s_{f+1} that have a transition between them in $\hat{M}_i$, but not in M. The 'Dead-end' states DE are the set of concrete states that are united under s_f and are reachable via paths that satisfy the counterexample. No single transition from DE can reach a concrete state in s_{f+1}, hence the name 'Dead-end'. The 'bad' states B are the set of states in s_f that can lead to concrete states in s_{f+1}. By definition, $DE \cap B = \emptyset$. The reason that the counterexample satisfies $\hat{M}_i$ is that DE and B are united under the same abstract state s_f. The refinement procedure in both [2] and [6] try to separate these two sets into different abstract states, thus eliminating the counterexample.

The technique in [5] splits each abstract state along the counterexample into two abstract states, one that corresponds to the continuous states that can be reached and the other one to the continuous states that cannot be reached along the counterexample path.

The refinement techniques in [1], [4] and [3] are all based on an analysis of conflict graphs that are built by the SAT solver when the formula is *unsatisfiable*. The unsatisfiable instance corresponds to a formula that conjoins the abstract counterexample with the concrete model. When the counterexample cannot be simulated on the concrete machine, this formula is unsatisfiable. By analyzing the conflict graph corresponding to the derivation of the empty clause, they identify the clauses and variables that contributed to the contradiction, and use this information to refine the model.
The articles include some experimental results comparing the methods. A good measure for the success of each algorithm (other than the total running time) is the size of the last abstracted model $\hat{M}_k$ and the number k of required iterations to reach this model. Clearly the size of $\hat{M}_k$ heavily depends on M and φ.

References

1. Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In Aagaard O'Leary, editor, *Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, lncs, Portland, Oregon, Nov 2002.
2. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proc. 12^{th} Intl. Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2000.
3. Edmund Clarke, Orna Grumberg, Muralidhar Talupur, and Dong Wang. High level verification of control intensive systems using predicate abstraction. In *Formal Methods and Models for Codesign (MEMOCODE'2003)*, Mont Saint-Michel, France, June 2003.
4. Edmund Clarke, Muralidhar Talupur, and Dong Wang. Sat based predicate abstraction for hardware verification. In *SAT'03*, S. Margherita Ligure - Portofino (Italy), May 2003.

Category	[2]	[6]	[5]
Model	discrete finite-state	discrete finite-state	discrete/continuous
Abstraction	Replacing Predicates with Boolean variables.	Making variables 'invisible', i.e. removing the logic that defines their allowed transitions, and referring to them as free inputs.	Continuous state-space of each discrete state (location) is partitioned into polyhedra.
Checking the counterexample	OBDDs	SAT	over-approximations of continuous dynamics
Refinement	Computing with OBDDs the full set of Dead-end and Bad states and separating them.	Sampling the set of Dead-end and Bad states and finding a small set of variables that separate them.	Split locations along the counterexample based on computed over-approximations of reachable states.

Category	[1]	[4]	[3]
Model	Discrete finite-state	Discrete finite-state	Discrete finite-state describing Control Intensive Systems
Abstraction	Same as [6]	Predicate abstraction	Predicate abstraction
Checking the counterexample	SAT	SAT	SAT
Refinement	Identifying variables that deduce the empty clause in the SAT based check of the counterexample.	Identifying variables corresponding to predicates that deduce the empty clause in the SAT based check of the counterexample.	Same as [4]

Fig. 1. A high-level comparison of the six approaches with respect to the various stages of the abstraction / refinement framework

5. E.M. Clarke, A. Fehnker, Z. Han, B. Krogh, O. Stursberg, and M. Theobald. Verification of hybrid systems based on counterexample-guided abstraction refinement. In *TACAS*, volume 2619 of *LNCS*. Springer, 2003.
6. E.M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction - refinement using ILP and machine learning techniques. In E. Brinksma and K.G. Larsen, editors, *Proc. 14^{th} Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 265–279, Copenhagen, Denmark, July 2002. Springer-Verlag.
7. R. Kurshan. *Computer aided verification of coordinating processes.* Princeton University Press, 1994.

Counterexample-guided Abstraction Refinement [$\star$]

Edmund Clarke[1], Orna Grumberg[2], Somesh Jha[1], Yuan Lu[1], and Helmut Veith[1,3]

[1] Carnegie Mellon University, Pittsburgh, USA [2] Technion, Haifa, Israel
[3] Vienna University of Technology, Austria

Abstract. We present an automatic iterative abstraction-refinement methodology in which the initial abstract model is generated by an automatic analysis of the control structures in the program to be verified. Abstract models may admit erroneous (or "spurious") counterexamples. We devise new symbolic techniques which analyze such counterexamples and refine the abstract model correspondingly. The refinement algorithm keeps the size of the abstract state space small due to the use of abstraction functions which distinguish many degrees of abstraction for each program variable. We describe an implementation of our methodology in NuSMV. Practical experiments including a large Fujitsu IP core design with about 500 latches and 10000 lines of SMV code confirm the effectiveness of our approach.

1 Introduction

The state explosion problem remains a major hurdle in applying model checking to large industrial designs. Abstraction is certainly the most important technique for handling this problem. In fact, it is essential for verifying designs of industrial complexity. Currently, abstraction is typically a manual process, often requiring considerable creativity. In order for model checking to be used more widely in industry, automatic techniques are needed for generating abstractions. In this paper, we describe an automatic abstraction technique for ACTL* specifications which is based on an analysis of the structure of formulas appearing in the program ($\mathrm{ACTL}^\star$ is a fragment of $\mathrm{CTL}^\star$ which only allows universal quantification over paths). In general, our technique computes an upper approximation of the original program. Thus, when a specification is true in the abstract model, it will also be true in the concrete design. However, if the specification is false in the abstract model, the counterexample may be the result of some behavior in the approximation which is not present in the original model. When this happens, it is necessary to refine the abstraction so that the behavior which caused the erroneous counterexample is eliminated. The main contribution of this paper is an efficient automatic refinement technique which uses information obtained from erroneous counterexamples. The refinement algorithm keeps the size of the abstract state

[$\star$] This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Max Kade Foundation. One of the authors is also supported by Austrian Science Fund Project N Z29-INF. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, or the United States Government.

space small due to the use of abstraction functions which distinguish many degrees of abstraction for each program variable. Practical experiments including a large Fujitsu IP core design with about 500 latches and 10000 lines of SMV code confirm the competitiveness of our implementation. Although our current implementation is based on NuSMV, it is in principle not limited to the input language of SMV and can be applied to other languages.

Our paper follows the general framework established by Clarke, Grumberg, and Long [10]. We assume that the reader has some familiarity with that framework. In our methodology, *atomic formulas* are automatically extracted from the program that describes the model. The atomic formulas are similar to the *predicates* used for abstraction by Graf and Saidi [13] and later in [11, 20]. However, instead of using the atomic formulas to generate an abstract global transition system, we use them to construct an explicit *abstraction function*. The abstraction function preserves logical relationships among the atomic formulas instead of treating them as independent propositions. The initial abstract model is constructed by adapting the *existential abstraction* techniques proposed in [8, 10] to our framework. Then, a traditional model checker is used to determine whether $\mathrm{ACTL}^{\star}$ properties hold in the abstract model. If the answer is yes, then the concrete model also satisfies the property. If the answer is no, then the model checker generates a counterexample. Since the abstract model has more behaviors than the concrete one, the abstract counterexample might not be valid. We say that such a counterexample is *spurious*. Such abstraction techniques are also known as false negative techniques.

In our methodology, we provide a new symbolic algorithm to determine whether an abstract counterexample is spurious. If the counterexample is not spurious, we report it to the user and stop. If the counterexample is spurious, the abstraction function must be refined to eliminate it. In our methodology, we identify the shortest prefix of the abstract counterexample that does not correspond to an actual trace in the concrete model. The last abstract state in this prefix is split into less abstract states so that the spurious counterexample is eliminated. Thus, a more refined abstraction function is obtained. Note that there may be many ways of splitting the abstract state; each determines a different refinement of the abstraction function. It is desirable to obtain the coarsest refinement which eliminates the counterexample because this corresponds to the *smallest* abstract model that is suitable for verification. We prove, however, that finding the coarsest refinement is NP-hard. Because of this, we use a polynomial-time algorithm which gives a suboptimal but sufficiently good refinement of the abstraction function. The applicability of our heuristic algorithm is confirmed by our experiments. Using the refined abstraction function obtained in this manner, a new abstract model is built and the entire process is repeated. Our methodology is complete for the fragment of $\mathrm{ACTL}^{\star}$ which has counterexamples that are either paths or loops, i.e., we are guaranteed to either find a valid counterexample or prove that the system satisfies the desired property. In principle, our methodology can be extended to all of $\mathrm{ACTL}^{\star}$.

Using counterexamples to refine abstract models has been investigated by a number of other researchers beginning with the *localization reduction* of Kurshan [14]. He models a concurrent system as a composition of L-processes $L_1, \ldots, L_n$ (L-processes are described in detail in [14]). The localization reduction is an iterative technique that starts with a small subset of relevant L-processes that are topologically close to the specification in the *variable dependency graph*. All other program variables are abstracted away with nondeterministic assignments. If the counterexample is found to be spurious, additional variables are added to eliminate the counterexample. The heuristic for selecting these variables also uses information from the variable dependency graph. Note that the localization reduction either leaves a variable unchanged or replaces it by a nondeterministic assignment. A similar approach has been described by Balarin in [2, 15]. In our approach, the abstraction functions exploit logical relationships among variables appearing in atomic formulas that occur in the control structure of the program. Moreover, the way we use abstraction functions makes it possible to distinguish many degrees of abstraction for each variable. Therefore, in the refinement step only very small and local changes to the abstraction functions are necessary and the abstract model remains comparatively small.

Another refinement technique has recently been proposed by Lind-Nielson and Andersen [17]. Their model checker uses upper and lower approximations in order to handle all of CTL. Their approximation techniques enable them to avoid rechecking the entire model after each refinement step while guaranteeing completeness. As in [2, 14] the variable dependency graph is used both to obtain the initial abstraction and in the refinement process. Variable abstraction is also performed in a similar manner. Therefore, our abstraction-refinement methodology relates to their technique in essentially the same way as it relates to the classical localization reduction.

A number of other papers [16, 18, 19] have proposed abstraction-refinement techniques for CTL model checking. However, these papers do not use counterexamples to refine the abstraction. We believe that the methods described in these papers are orthogonal to our technique and may even be combined with ours in order to achieve better performance. A recent technique proposed by Govindaraju and Dill [12] may be a starting point in this direction, since it also tries to identify the first spurious state in an abstract counterexample. It randomly chooses a concrete state corresponding to the first spurious state and tries to construct a real counterexample starting with the image of this state under the transition relation. The paper only talks about safety properties and path counterexamples. It does not describe how to check liveness properties with cyclic counterexamples. Furthermore, our method does not use random choice to extend the counterexample; instead it analyzes the cause of the spurious counterexample and uses this information to guide the refinement process. A more detailed comparison with related work will be given in the full version

Summarizing, our technique has a number of advantages over previous work:

(i) The technique is complete for an important fragment of ACTL*.

(ii) The initial abstraction and the refinement steps are efficient and entirely automatic. All algorithms are symbolic.

(iii) In comparison to methods like the localization reduction, we distinguish more degrees of abstraction for each variable. Thus, the changes in the refinement are potentially finer in our approach.

(iv) The refinement procedure is guaranteed to eliminate spurious counterexamples while keeping the state space of the abstract model small.

We have implemented our new methodology in NuSMV [6] and applied it to a number of benchmark designs [6]. In addition we have used it to debug a large IP core being developed at Fujitsu [1]. The design has about 350 symbolic variables which correspond to about 500 latches. Before using our methodology, we implemented the *cone of influence* reduction [8] in NuSMV to enhance its ability to check large models. Neither our enhanced version of NuSMV nor the recent version of SMV developed by Yang [23] were able to verify the Fujitsu IP core design. However, by using our new technique, we were able to find a subtle error in the design. Our program automatically abstracted 144 symbolic variables and performed three refinement steps. Currently, we are evaluating the methodology on other complex industrial designs.

The paper is organized as follows: Section 2 gives the basic definitions and terminology used throughout the paper. A general overview of our methodology is given in Section 3. Detailed descriptions of our abstraction-refinement algorithms are provided in Section 4. Performance improvements for the implementation are described in Section 5. Experimental results are presented in Section 6. Future research is discussed in Section 7.

2 Preliminaries

A *program* P has a finite set of variables $V = \{v_1, \cdots, v_n\}$, where each variable v_i has an associated finite domain D_{v_i}. The set of all possible states for program P is $D_{v_1} \times \cdots D_{v_n}$ which we denote by D. *Expressions* are built from variables in V, constants in D_{v_i}, and function symbols in the usual way, e.g. $v_1 + 3$. *Atomic formulas* are constructed from expressions and relation symbols, e.g. $v_1 + 3 < 5$. Similarly, *predicates* are composed of atomic formulas using negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). Given a predicate p, $\mathrm{Atoms}(p)$ is the set of atomic formulas occurring in it. Let p be a predicate containing variables from V, and $d = (d_1, \ldots, d_n)$ be an element from D. Then we write $d \models p$ when the predicate obtained by replacing each occurrence of the variable v_i in p by the constant d_i evaluates to true.

Each variable v_i in the program has an associated *transition block*, which defines both the initial value and the transition relation for the variable v_i. An example of a transition block for the variable v_i is shown in Figure 1, where $I_i \subseteq D_{v_i}$ is the initial

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

Fig. 1. A generic transition block and a typical example

expression for the variable v_i, each condition C_i^j is a predicate, and A_i^j is an expression.

The semantics of the transition block is similar to the semantics of the **case** statement in the modeling language of SMV, i.e., find the least j such that in the current state condition C_i^j is true and assign the value of the expression A_i^j to the variable v_i in the next state.

We assume that the specifications are written in a fragment of $\mathrm{CTL}^\star$ called $\mathrm{ACTL}^\star$ (see [10]). Assume that we are given an $\mathrm{ACTL}^\star$ specification φ, and a program P. For each transition block B_i let $\mathrm{Atoms}(B_i)$ be the set of atomic formulas that appear in the conditions. Let $\mathrm{Atoms}(\varphi)$ be the set of atomic formulas appearing in the specification φ. $\mathrm{Atoms}(P)$ is the set of atomic formulas that appear in the specification or in the conditions of the transition blocks.

Each program P naturally corresponds to a labeled *Kripke structure* $M = (S, I, R, L)$, where $S = D$ is the set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \to 2^{\mathrm{Atoms}(P)}$ is a labelling given by $L(d) = \{f \in \mathrm{Atoms}(P) \mid d \models f\}$. Translating a program into a Kripke structure is straightforward and will not be described here.

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

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

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

Assume that we are given a program P and an abstraction function h for P. The *abstract Kripke structure* $\widehat{M} = (\widehat{S}, \widehat{I}, \widehat{R}, \widehat{L})$ corresponding to the abstraction function h is defined as follows:

1. $\widehat{S}$ is the abstract domain $\widehat{D}$.
2. $\widehat{I}(\widehat{d})$ iff $\exists d(h(d) = \widehat{d} \wedge I(d))$.
3. $\widehat{R}(\widehat{d_1}, \widehat{d_2})$ iff $\exists d_1 \exists d_2(h(d_1) = \widehat{d_1} \wedge h(d_2) = \widehat{d_2} \wedge R(d_1, d_2))$.
4. $\widehat{L}(\widehat{d}) = \bigcup_{h(d)=\widehat{d}} L(d)$.   (This definition will be justified in Theorem 1.)

This abstraction technique is called *existential abstraction* [8]. An atomic formula f *respects* an abstraction function h if for all d and d' in the domain D, $(d \equiv d') \Rightarrow (d \models f \Leftrightarrow d' \models f)$. Let $\widehat{d}$ be an abstract state. $\widehat{L}(\widehat{d})$ is *consistent*, if all concrete states corresponding to $\widehat{d}$ satisfy all labels in $\widehat{L}(\widehat{d})$, i.e., for all $d \in h^{-1}(\widehat{d})$ it holds that $d \models \bigwedge_{f \in \widehat{L}(\widehat{d})} f$.

Theorem 1. *Let h be an abstraction and φ be an $\mathrm{ACTL}^\star$ specification where the atomic subformulas respect h. Then the following holds: (i) $\widehat{L}(\widehat{d})$ is consistent for all abstract states $\widehat{d}$ in $\widehat{M}$; (ii) $\widehat{M} \models \varphi \;\Rightarrow\; M \models \varphi$.*

In other words, correctness of the abstract model implies correctness of the concrete model. On the other hand, if the abstract model invalidates an $\mathrm{ACTL}^\star$ specification, i.e., $\widehat{M} \not\models \varphi$, *the actual model may still satisfy the specification.*

Example 1. Assume that for a traffic light controller (see Figure 2), we want to prove $\psi = \mathbf{AG}\,\mathbf{AF}(state = red)$ using the abstraction function $h(red) = red$ and $h(green) = h(yellow) = go$. It is easy to see that $M \models \psi$ while $\widehat{M} \not\models \psi$. There exists an infinite trace $\langle red, go, go, \ldots \rangle$ that invalidates the specification.

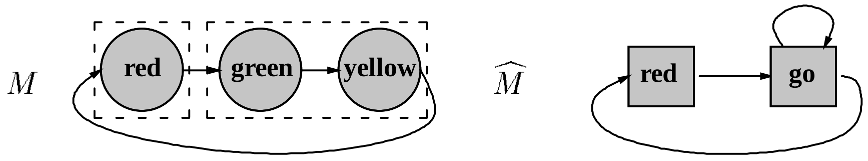


Fig. 2. Abstraction of a Traffic Light.

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

When the set of possible states is given as the product $D_1 \times \cdots D_n$ of smaller domains, an abstraction h can be described by surjections $h_i : D_i \to \widehat{D_i}$, such that $h(d_1, \ldots, d_n)$ is equal to $(h_1(d_1), \ldots, h_n(d_n))$, and $\widehat{D}$ is equal to $\widehat{D_1} \times \cdots \widehat{D_n}$. In this case, we write $h = (h_1, \ldots, h_n)$. The equivalence relations $\equiv_i$ corresponding to the individual surjections h_i induce an equivalence relation $\equiv$ over the entire domain $D = D_1 \times \cdots \times D_n$ in the obvious manner:

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

In previous work on existential abstraction [10], abstractions were defined for each variable domain, i.e., D_i in the above paragraph was chosen to be D_{v_i}, where D_{v_i} is the set of possible values for variable v_i. Unfortunately, many abstraction functions h can not be described in this simple manner. For example, let $D = \{0,1,2\} \times \{0,1,2\}$, and $\widehat{D} = \{0,1\} \times \{0,1\}$. Then there are $4^9 = 262144$ functions h from D to $\widehat{D}$. Next, consider $h = (h_1, h_2)$. Since there are $2^3 = 8$ functions from $\{0,1,2\}$ to $\{0,1\}$, there are only 64 functions of this form from D to $\widehat{D}$.

In this paper, we define abstraction functions in a different way. We partition the set V of variables into sets of related variables called *variable clusters* $VC_1, \ldots, VC_m$, where each variable cluster VC_i has an associated domain $D_{VC_i} := \prod_{v \in VC_i} D_v$. Consequently, $D = D_{VC_1} \times \cdots D_{VC_m}$. We define abstraction functions as surjections on the domains D_{VC_i}, i.e., D_i in the above paragraph is equal to D_{VC_i}. Thus, the notion of abstraction used in this paper is more general than the one used in [10].

3 Overview

For a program P and an ACTL* formula φ, our goal is to check whether the Kripke structure M corresponding to P satisfies φ. Our methodology consists of the following steps.

1. *Generate the initial abstraction:* We generate an initial abstraction h by examining the transition blocks corresponding to the variables of the program. We consider the conditions used in the **case** statements and construct variable clusters for variables which interfere with each other via these conditions. Details can be found in Section 4.1.
2. *Model-check the abstract structure:* Let $\widehat{M}$ be the abstract Kripke structure corresponding to the abstraction h. We check whether $\widehat{M} \models \varphi$. If the check is affirmative, then we can conclude that $M \models \varphi$ (see Theorem 1). Suppose the check reveals that there is a counterexample $\widehat{T}$. We ascertain whether $\widehat{T}$ is an actual counterexample, i.e., a counterexample in the unabstracted structure M. If $\widehat{T}$ turns out to be an actual counterexample, we report it to the user, otherwise $\widehat{T}$ is a spurious counterexample, and we proceed to step 3.
3. *Refine the abstraction:* We refine the abstraction function h by partitioning a *single equivalence class* of $\equiv$ so that after the refinement the abstract structure $\widehat{M}$ corresponding to the refined abstraction function does not admit the spurious counterexample $\widehat{T}$. We will discuss partitioning algorithms for this purpose in Section 4.3. After refining the abstraction function, we return to step 2.

4 The Abstraction-Refinement Framework

4.1 Generating the Initial Abstraction

Assume that we are given a program P with n variables $\{v_1, \cdots, v_n\}$. Given an atomic formula f, let $var(f)$ be the set of variables appearing in f, e.g., $var(x = y)$ is $\{x, y\}$. Given a set of atomic formulas U, $var(U)$ equals $\bigcup_{f \in U} var(f)$. In general, for any syntactic entity X, $var(X)$ will be the set of variables appearing in X. We say that two atomic formulas f_1 and f_2 *interfere* iff $var(f_1) \cap var(f_2) \neq \emptyset$. Let $\equiv_I$ be the equivalence relation on $\mathrm{Atoms}(P)$ that is the reflexive, transitive closure of the interference relation. The equivalence class of an atomic formula $f \in \mathrm{Atoms}(P)$ is called the *formula cluster* of f and is denoted by $[f]$. Let f_1 and f_2 be two atomic formulas. Then $var(f_1) \cap var(f_2) \neq \emptyset$ implies that $[f_1] = [f_2]$. In other words, a variable v_i cannot appear in formulas that belong to two different formula clusters. Moreover, the formula clusters induce an equivalence relation $\equiv_V$ on the set of variables V in the following way:

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

The equivalence classes of $\equiv_V$ are called *variable clusters*. For instance, consider a formula cluster $FC_i = \{v_1 > 3, v_1 = v_2\}$. The corresponding variable cluster is $VC_i = \{v_1, v_2\}$. Let $\{FC_1, \ldots, FC_m\}$ be the set of formula clusters and $\{VC_1, \ldots, VC_m\}$ the set of corresponding variable clusters. We construct the initial abstraction $h = (h_1, \ldots, h_m)$ as follows. For each h_i, we set $D_{VC_i} = \prod_{v \in VC_i} D_v$, i.e., D_{VC_i} is the domain corresponding to the variable cluster VC_i. Since the variable clusters form a partition of the set of variables V, it follows that $D = D_{VC_1} \times \cdots D_{VC_m}$. For each variable cluster $VC_i = \{v_{i_1}, \ldots, v_{i_k}\}$, the corresponding abstraction h_i is defined on D_{VC_i} as follows.

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

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

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

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

The domain $\{\mathrm{TRUE}, \mathrm{FALSE}\}$ has two equivalence classes – one containing FALSE and the other TRUE. Therefore, we define two abstraction functions $h_1 : \{0, 1, 2\}^2 \rightarrow \{0, 1, 2, 3, 4\}$ and $h_2 : \{\mathrm{TRUE}, \mathrm{FALSE}\} \rightarrow \{\mathrm{TRUE}, \mathrm{FALSE}\}$. The first function h_1 is given by $h_1(0, 0) = h_1(1, 1) = 0$, $h_1(0, 1) = 1$, $h_1(0, 2) = h_1(1, 2) = 2$, $h_1(1, 0) = h_1(2, 0) = h_1(2, 1) = 3$, $h_1(2, 2) = 4$. The second function h_2 is just the identity function, i.e., $h_2(reset) = reset$. Given the abstraction functions, we use the standard existential abstraction techniques to compute the abstract model.

4.2 Model Checking the Abstract Model

Given an $\mathrm{ACTL}^\star$ specification φ, an abstraction function h (assume that φ respects h), and a program P with a finite set of variables $V = \{v_1, \cdots, v_n\}$, let $\widehat{M}$ be the abstract Kripke structure corresponding to the abstraction function h. We use standard symbolic model checking procedures to determine whether $\widehat{M}$ satisfies the specification φ. If it does, then by Theorem 1 we can conclude that the original Kripke structure also satisfies φ. Otherwise, assume that the model checker produces a counterexample $\widehat{T}$ corresponding to the abstract model $\widehat{M}$. In the rest of this section, we will focus on counterexamples which are either *(finite) paths* or *loops*.

Identification of Spurious Path Counterexamples First, we will tackle the case when the counterexample $\widehat{T}$ is a path $\langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$. Given an abstract state $\widehat{s}$, the set of concrete states s such that $h(s) = \widehat{s}$ is denoted by $h^{-1}(\widehat{s})$, i.e., $h^{-1}(\widehat{s}) = \{s | h(s) = \widehat{s}\}$.

We extend h^{-1} to sequences in the following way: $h^{-1}(\widehat{T})$ is the set of concrete paths given by the following expression

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

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

Lemma 1. *The following are equivalent:*

(i) The path $\widehat{T}$ corresponds to a concrete counterexample.
(ii) The set of concrete paths $h^{-1}(\widehat{T})$ is non-empty.
(iii) For all $1 \leq i \leq n$, $S_i \neq \emptyset$.

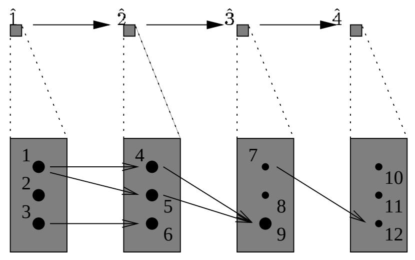


Fig. 3. An abstract counterexample

```
Algorithm SplitPATH
$S := h^{-1}(\widehat{s_1}) \cap I$
$j := 1$
while ($S \neq \emptyset$ and $j < n$) {
        $j := j + 1$
        $S_{\text{prev}} := S$
        $S := Img(S, R) \cap h^{-1}(\widehat{s_j})$ }
if $S \neq \emptyset$ then output counterexample
else output j, $S_{\text{prev}}$
```

Fig. 4. SplitPATH checks spurious path.

Example 3. Consider a program with only one variable with domain $D = \{1, \cdots, 12\}$. Assume that the abstraction function h maps $x \in D$ to $\lfloor (x-1)/3 \rfloor + 1$. There are four abstract states corresponding to the equivalence classes $\{1, 2, 3\}$, $\{4, 5, 6\}$, $\{7, 8, 9\}$, and $\{10, 11, 12\}$. We call these abstract states $\widehat{1}$, $\widehat{2}$, $\widehat{3}$, and $\widehat{4}$. The transitions between states in the concrete model are indicated by the arrows in Figure 3; small dots denote non-reachable states. Suppose that we obtain an abstract counterexample $\widehat{T} = \langle \widehat{1}, \widehat{2}, \widehat{3}, \widehat{4} \rangle$. It is easy to see that $\widehat{T}$ is spurious. Using the terminology of Lemma 1, we have $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Notice that S_4 and therefore $Img(S_3, R)$ are both empty.

It follows from Lemma 1 that if $h^{-1}(\widehat{T})$ is empty (i.e., if the counterexample $\widehat{T}$ is spurious), then there exists a minimal i ($2 \leq i \leq n$) such that $S_i = \emptyset$. The symbolic Algorithm **SplitPATH** in Figure 4 computes this number and the set of states in

S_{i-1}. In this case, we proceed to the refinement step (see Section 4.3). On the other hand, if the conditions stated in Lemma 1 are true, then **SplitPATH** will report a "real" counterexample and we can stop.

Identification of Spurious Loop Counterexamples Now we consider the case when the counterexample $\widehat{T}$ includes a loop, which we write as $\langle \widehat{s_1}, \cdots, \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, \cdots, \widehat{s_n} \rangle^{\omega}$. The loop starts at the abstract state $\widehat{s_{i+1}}$ and ends at $\widehat{s_n}$. Since this case is more complicated than the path counterexamples, we first present an example in which some of the typical situations occur.

Example 4. We consider a loop $\langle \widehat{s_1} \rangle \langle \widehat{s_2}, \widehat{s_3} \rangle^{\omega}$ as shown in Figure 5. In order to find out if the abstract loop corresponds to concrete loops, we unwind the counterexample as demonstrated in the figure. There are two situations where cycles occur. In the figure,

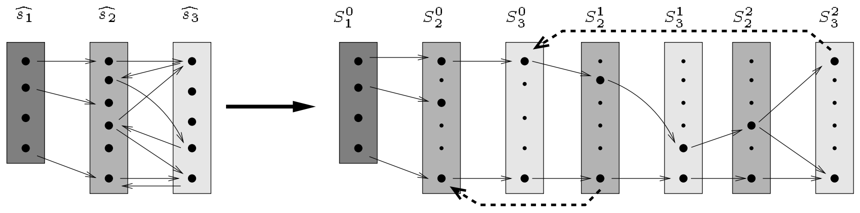


Fig. 5. A loop counterexample, and its unwinding.

for each of these situations, an example cycle (the first one occurring) is indicated by a fat dashed arrow. We make the following important observations: (i) A given abstract loop may correspond to several concrete loops of *different size*. (ii) Each of these loops may start at different stages of the unwinding. (iii) The unwinding eventually becomes periodic (in our case $S_3^0 = S_3^2$), but only after several stages of the unwinding. The size of the period is the least common multiple of the size of the individual loops, and thus, in general *exponential*.

We conclude from the example that a naive algorithm may have exponential time complexity due to an exponential number of loop unwindings. The following surprising result shows that for $\widehat{T} = \langle \widehat{s_1}, \cdots, \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, \cdots, \widehat{s_n} \rangle^{\omega}$, the number of unwindings can be bounded by $min = \min_{i+1 \leq j \leq n} |h^{-1}(\widehat{s_j})|$, i.e., the number of unwindings is at most the number of concrete states for any abstract state in the loop. Let $\widehat{T}_{\text{unwind}}$ denote this unwinded loop counterexample, i.e., the finite abstract path $\langle \widehat{s_1}, \ldots, \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, \ldots, \widehat{s_n} \rangle^{min}$. Then the following theorem holds.

Theorem 2. *The following are equivalent: (i) $\widehat{T}$ corresponds to a concrete counterexample. (ii) $h_{\text{path}}^{-1}(\widehat{T}_{\text{unwind}})$ is not empty.*

It can be seen from Example 4 that loop counterexamples are combinatorially more complicated than path counterexamples. Therefore, the proof of Theorem 2 is not immediate; for details, we refer to [7]. We conclude from Theorem 2 that the Algorithm

SplitPATH can be used to analyze abstract loop counterexamples with minor modifications. For easy reference we shall refer to this algorithm as **SplitLOOP**.

4.3 Refining the Abstraction

First, we will consider the case when the counterexample $\widehat{T} = \langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$ is a path. Since $\widehat{T}$ does not correspond to a real counterexample, by Lemma 1 (iii) there exists a set $S_i \subseteq h^{-1}(\widehat{s_i})$ with $1 \leq i < n$ such that $Img(S_i, R) \cap h^{-1}(\widehat{s_{i+1}}) = \emptyset$ and S_i is reachable from initial state set $h^{-1}(\widehat{s_1}) \cap I$. Since there is a transition from $\widehat{s_i}$ to $\widehat{s_{i+1}}$ in the abstract model, there is at least one transition from a state in $h^{-1}(\widehat{s_i})$ to a state in $h^{-1}(\widehat{s_{i+1}})$ even though there is no transition from S_i to $h^{-1}(\widehat{s_{i+1}})$. We partition $h^{-1}(\widehat{s_i})$ into three subsets $S_{i,0}$, $S_{i,1}$, and $S_{i,x}$ as follows (compare Figure 6):

$$\begin{aligned}
&S_{i,0} = S_i \\
&S_{i,1} = \{s \in h^{-1}(\widehat{s_i}) | \exists s' \in h^{-1}(\widehat{s_{i+1}}).R(s, s')\} \\
&S_{i,x} = h^{-1}(\widehat{s_i}) \setminus (S_{i,0} \cup S_{i,1}).
\end{aligned}$$

Intuitively, $S_{i,0}$ denotes the set of states in $h^{-1}(\widehat{s_i})$ that are reachable from initial states. $S_{i,1}$ denotes the set of states in $h^{-1}(\widehat{s_i})$ that are not reachable from initial states, but have at least one transition to some state in $h^{-1}(\widehat{s_{i+1}})$. The set $S_{i,1}$ cannot be empty since we know that there is a transition from $h^{-1}(\widehat{s_i})$ to $h^{-1}(\widehat{s_{i+1}})$. $S_{i,x}$ denotes the set of states that are not reachable from initial states, and do not have a transition to a state in $h^{-1}(\widehat{s_{i+1}})$. For illustration, consider again the example in Figure 3. Note that $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Using the notation introduced above, we have $S_{3,0} = \{9\}$, $S_{3,1} = \{7\}$, and $S_{3,x} = \{8\}$. Since $S_{i,1}$ is not empty, there is a spurious transition $\widehat{s_i} \rightarrow \widehat{s_{i+1}}$. This causes the spurious counterexample $\widehat{T}$. Hence in order to refine the abstraction h so that the new model does not allow $\widehat{T}$, we need a refined abstraction function which separates the two sets $S_{i,0}$ and $S_{i,1}$, i.e., we need an abstraction function, in which no abstract state simultaneously contains states from $S_{i,0}$ and from $S_{i,1}$.

It is natural to describe the needed refinement in terms of equivalence relations: Recall that $h^{-1}(\widehat{s})$ is an equivalence class of $\equiv$ which has the form $E_1 \times \cdots \times E_m$, where each E_i is an equivalence class of $\equiv_i$. Thus, the refinement $\equiv'$ of $\equiv$ is obtained by partitioning the equivalence classes E_j into subclasses, which amounts to refining the equivalence relations $\equiv_j$. The *size of the refinement* is the number of new equivalence classes. Ideally, we would like to find the coarsest refinement that separates the two sets, i.e., the separating refinement with the smallest size. We can show however that this is computationally intractable.

Theorem 3. *(i) The problem of finding the coarsest refinement is NP-hard; (ii) when $S_{i,x} = \emptyset$, the problem can be solved in polynomial time.*

We find that the previously known poblem PARTITION INTO CLIQUES can be reduced to the coarsest refinement problem. The proof is omitted due to space restrictions. On the other hand, we describe a polynomial time algorithm **PolyRefine** corresponding to case (ii) of Theorem 3 in Figure 7. Let P_j^+, P_j^- be two projection functions, such that for $s = (d_1, \ldots, d_m)$, $P_j^+(s) = d_j$ and

$P_j^-(s) = (d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$. Then $proj(S_{i,0}, j, a)$ denotes the *projection* set $\{P_j^-(s) | P_j^+(s) = a, s \in S_{i,0}\}$. Intuitively, the condition $proj(S_{i,0}, j, a) \neq proj(S_{i,0}, j, b)$ in the algorithm means that there exists $(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \in proj(S_{i,0}, j, a)$ and $(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \notin proj(S_{i,0}, j, b)$. According to the definition of $proj(S_{i,0}, j, a)$, $s_1 = (d_1, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in S_{i,0}$ and $s_2 = (d_1, \ldots, d_{j-1}, b, d_{j+1}, \ldots, d_m) \notin S_{i,0}$, i.e., $s_2 \in S_{i,1}$. Note that s_1 and s_2 are only different at j-th component. Hence, the only way to separate s_1 and s_2 into different equivalence classes is that a and b have to be in different equivalence classes of $\equiv'_j$, i.e., $a \not\equiv'_j b$.

Lemma 2. *When $S_{i,x} = \emptyset$, the relation $\equiv'_j$ computed by* **PolyRefine** *is an equivalence relation which refines $\equiv_j$ and separates $S_{i,0}$ and $S_{i,1}$. Furthermore, the equivalence relation $\equiv'_j$ is the coarsest refinement of $\equiv_j$.*

Note that in symbolic presentation, the projection operation $proj(S_{i,0}, j, a)$ amounts to computing a generalized cofactor, which can be easily done by standard BDD methods. Given a function $f : D \to \{0,1\}$, a generalized cofactor of f with respect to $g = (\bigwedge_{k=p}^{q} x_k = d_k)$ is the function $f_g = f(x_1, \ldots, x_{p-1}, d_p, \ldots, d_q, x_{q+1}, \ldots, x_n)$. In other words, f_g is the projection of f with respect to g. Symbolically, the set $S_{i,0}$ is represented by a function $f_{S_{i,0}} : D \to \{0,1\}$, and therefore, the projection $proj(S_{i,0}, j, a)$ of $S_{i,0}$ to value a of the jth component corresponds to a cofactor of $f_{S_{i,0}}$.

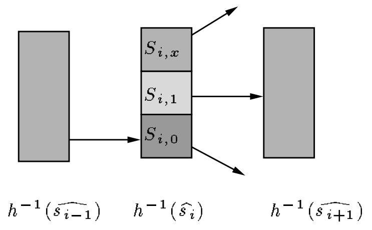


Fig. 6. Three sets $S_{i,0}$, $S_{i,1}$, and $S_{i,x}$

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

Fig. 7. The algorithm **PolyRefine**

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

Since according to Theorem 2, the algorithm **SplitLOOP** for loop counterexamples works analogously as **SplitPATH**, the refinement procedure for spurious loop counterexamples works analogously, too. Details are omitted due to space restrictions. Our refinement procedure continues to refine the abstraction function by partitioning equivalence classes until a real counterexample is found, or the $\mathrm{ACTL}^\star$ property is verified. The partitioning procedure is guaranteed to terminate since each equivalence class must contain at least one element. Thus, our method is complete.

Theorem 4. *Given a model M and an $\mathrm{ACTL}^\star$ specification φ whose counterexample is either path or loop, our algorithm will find a model $\widehat{M}$ such that $\widehat{M} \models \varphi \Leftrightarrow M \models \varphi$.*

5 Performance Improvements

The symbolic methods described in Section 4 can be directly implemented using BDDs. Our implementation uses additional heuristics which are outlined in this section. For details, we refer to our technical report [7].

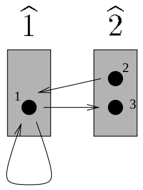


Fig. 8. A spurious loop counterexample $\langle\widehat{1},\widehat{2}\rangle^\omega$

Two-phase Refinement Algorithms. Consider the spurious loop counterexample $\widehat{T} = \langle\widehat{1},\widehat{2}\rangle^\omega$ of Figure 8. Although $\widehat{T}$ is spurious, the concrete states involved in the example contain an infinite path $\langle 1, 1, \ldots\rangle$ which is a potential counterexample. Since we know that our method is complete, such cases could be ignored. Due to practical performance considerations, however, we came to the conclusion that the relatively small effort to detect additional counterexamples is justified as a valuable heuristic. For a general loop counterexample $\widehat{T} = \langle\widehat{s}_1, \ldots, \widehat{s}_i\rangle\langle\widehat{s}_{i+1}, \ldots, \widehat{s}_n\rangle^\omega$, we therefore proceed in two phases:
(i) We restrict the model to the state space $S_{\mathrm{local}} := (\bigcup_{1\le i\le n} h^{-1}(\widehat{s}_i))$ of the counterexample and use the standard fixpoint computation for temporal formulas (see e.g. [8]) to check the property on the Kripke structure restricted to S_{local}. If a concrete counterexample is found, then the algorithm terminates.
(ii) If no counterexample is found, we use **SplitLOOP** and **PolyRefine** to compute a refinement as described above.
This two-phase algorithm is slightly slower than the original one if we do not find a concrete counterexample; in many cases however, it can speed up the search for a concrete counterexample. An analogous two phase approach is used for finite path counterexamples.

Approximation. Despite the use of partitioned transition relations it is often infeasible to compute the total transition relation of the model M [8]. Therefore, the abstract model $\widehat{M}$ cannot be computed from M directly. In previous work [2, 10], a method which we call *early approximation* has been introduced: first, abstraction is applied to the BDD representation of each transition block and then the BDDs for the partitioned transition relation are built from the already abstracted BDDs for the transition blocks. The disadvantage of early approximation is that it *over-approximates* the abstract model

$\widehat{M}$ [9]. In our approach, a heuristic individually determines for each variable cluster VC_i, if early approximation should be applied or if the abstraction function should be applied in an exact manner. Our method has the advantage that it balances overapproximation and memory usage. Moreover, the overall method presented in our paper remains complete with this approximation.

Abstractions For Distant Variables. In addition to the methods of Section 4.1, we completely abstract variables whose distance from the specification in the *variable dependency graph* is greater than a user-defined constant. Note that the variable dependency graph is also used for this purpose in the localization reduction [2, 14, 17] in a similar way. However, the refinement process of the localization reduction [14] can only turn a completely abstracted variable into a completely unabstracted variable, while our method uses intermediate abstraction functions.

6 Experimental Results

We have implemented our methodology in NuSMV [6] which uses the CUDD package [21] for symbolic representation. We performed two sets of experiments. One set is on five benchmark designs. The other was performed on an industrial design of a multimedia processor from Fujitsu [1]. All the experiments were carried out on a 200MHz PentiumPro PC with 1GB RAM memory using Linux.

The first benchmark designs are publicly available. The PCI example is extracted from [5]. The results for these designs are listed in the table.

Design	#Var	#Prop	**NuSMV+COI**				**NuSMV+ABS**			
			#COI	Time	$\lvert TR\rvert$	$\lvert MC\rvert$	#ABS	Time	$\lvert TR\rvert$	$\lvert MC\rvert$
gigamax	10(16)	1	0	0.3	8346	1822	9	0.2	13151	816
guidance	40(55)	8	30	35	140409	30467	34-39	30	147823	10670
p-queue	12(37)	1	4	0.5	51651	1155	5	0.4	52472	1114
waterpress	6(21)	4	0-1	273	34838	129595	4	170	38715	3335
PCI bus	50(89)	10	4	2343	121803	926443	12-13	546	160129	350226

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

Although our approach in one case uses 50% more memory than the traditional cone of influence reduction to *build* the abstract transition relation, it requires one magnitude less memory during *model checking*. This is an important achievement since the model checking process is the most difficult task in verifying large designs. More significant improvement is further demonstrated by the Fujitsu IP core design.

The Fujitsu IP core design is a multimedia assist (MMA-ASIC) processor [1]. The design is a system-on-a-chip that consists of a co-processor for multimedia instructions, a graphic display controller, peripheral I/O units, and five bus bridges. The RTL implementation of MM-ASIC is described in about 61,500 lines of Verilog-HDL code. After manual abstraction by engineers from Fujitsu in [22], there still remain about 10,600 lines of code with roughly 500 registers. We translated this abstracted Verilog code into 9,500 lines of SMV code. In [22], the authors verified this design using a ”navigated” model checking algorithm in which state traversal is restricted by navigation conditions provided by the user. Therefore, their methodology is not complete, i.e., it may fail to prove the correctness even if the property is true. Moreover, the navigation conditions are usually not automatically generated.

In order to compare our model checker to others, we tried to verify this design using two state-of-the-art model checkers - Yang's SMV [23] and NuSMV [6]. We implemented the cone of influence reduction for NuSMV, but not for Yang's SMV. Both NuSMV+COI and Yang's SMV failed to verify the design. On the other hand, our system abstracted 144 symbolic variables and with three refinement steps, successfully verified the design, and found a bug which has not been discovered before.

7 Conclusion and Future Work

We have presented a novel abstraction refinement methodology for symbolic model checking. The advantages of our methodology have been demonstrated by experimental results. We believe that our technique is general enough to be adapted for other forms of abstraction. There are many interesting avenues for future research. First, we want to find efficient approximation algorithms for the NP-complete separation problem encountered during the refinement step. Moreover, in a recent paper [4], the fragment of ACTL* that admits “trace”-like counterexamples (of a potentially more complicated structure than paths and loops) has been characterized; we plan to extend our refinement algorithm to this language. Since the symbolic methods described in this paper are not tied to representation by BDDs, we will also investigate how they can be applied to recent work on symbolic model checking without BDDs [3]. We are currently applying our technique to verify other large examples.

References

1. Fujitsu aims media processor at DVD. *MicroProcessor Report*, pages 11–13, 1996.
2. F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer-Aided Verification*, volume 697 of *LNCS*, pages 29–40, 1993.
3. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference*, pages 317–320, 1999.
4. F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. On ACTL formulas having deterministic counterexamples. Technical report, Vienna University of Technology, 1999. available at http://www.kr.tuwien.ac.at/research/reports/index.html.

5. P. Chauhan, E. Clarke, Y. Lu, and D. Wang. Verifying IP-core based System-On-Chip design. In *IEEE ASIC*, September 1999.
6. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Software Tools for Technology Transfer*, 1998.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. Technical Report CMU-CS-00-103, Computer Science, Carnegie Mellon University, 2000.
8. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Publishers, 1999.
9. E. Clarke, S. Jha, Y. Lu, and D. Wang. Abstract BDDs: a technique for using abstraction in model checking. In *Correct Hardware Design and Verifi cation Methods*, volume 1703 of *LNCS*, pages 172–186, 1999.
10. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5):1512–1542, September 1994.
11. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Computer-Aided Verifi cation*, volume 1633 of *LNCS*, pages 160–171. Springer Verlag, July 1999.
12. Shankar G. Govindaraju and David L. Dill. Verification by approximate forward and backward reachability. In *Proceedings of International Conference on Computer-Aided Design*, November 1998.
13. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer-Aided Verifi cation*, volume 1254 of *LNCS*, pages 72–83, June 1997.
14. R. P. Kurshan. *Computer-Aided Verifi cation of Coordinating Processes*. Princeton University Press, 1994.
15. Y. Lakhnech. personal communication. 2000.
16. W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 76–81, November 1996.
17. J. Lind-Nielsen and H. R. Andersen. Stepwise CTL model checking of state/event systems. In *Computer-Aided Verifi cation*, volume 1633 of *LNCS*, pages 316–327. Springer Verlag, 1999.
18. A. Pardo. *Automatic Abstraction Techniques for Formal Verifi cation of Digital Systems*. PhD thesis, University of Colorado at Boulder, Dept. of Computer Science, August 1997.
19. A. Pardo and G.D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference*, pages 457–462, 1998.
20. H. Saidi and N. Shankar. Abstract and model checking while you prove. In *Computer-Aided Verifi cation*, number 1633 in LNCS, pages 443–454, July 1999.
21. F. Somenzi. CUDD: CU decision diagram package. Technical report, University of Colorado at Boulder, 1997.
22. K. Takayama, T. Satoh, T. Nakata, and F. Hirose. An approach to verify a large scale system-on-chip using symbolic model checking. In *International Conference of Computer Design*, pages 308–313, 1998.
23. B. Yang et al. A performance study of BDD-based model checking. In *Formal Methods in Computer-Aided Design*, volume 1522 of *LNCS*. Springer Verlag, 1998.

SAT based Abstraction-Refinement using ILP and Machine Learning Techniques [⋆]

Edmund Clarke[1] Anubhav Gupta[1] James Kukula[2] Ofer Strichman[1]

1. Computer Science, Carnegie Mellon University, Pittsburgh, PA
`emc|anubhav|ofers@cs.cmu.edu`
2. Synopsys, Beaverton, OR. `kukula@synopsys.com`

Abstract. We describe new techniques for model checking in the counterexample guided abstraction/refinement framework. The abstraction phase 'hides' the logic of various variables, hence considering them as inputs. This type of abstraction may lead to 'spurious' counterexamples, i.e. traces that can not be simulated on the original (concrete) machine. We check whether a counterexample is real or spurious with a SAT checker. We then use a combination of Integer Linear Programming (ILP) and machine learning techniques for refining the abstraction based on the counterexample. The process is repeated until either a real counterexample is found or the property is verified. We have implemented these techniques on top of the model checker NuSMV and the SAT solver Chaff. Experimental results prove the viability of these new techniques.

1 Introduction

While state of the art model checkers can verify circuits with several hundred latches, many industrial circuits are at least an order of magnitude larger. Various conservative abstraction techniques can be used to bridge this gap. Such abstraction techniques must preserve all the behaviors of the concrete system, but may introduce behaviors that are not present originally. Thus, if a universal property (i.e. an ACTL* property) is true in the abstract system, it will also be true in the concrete system. On the other hand, if a universal property is false in the abstract system, it may still be true in the concrete system. In this case, none of the behaviors that violate the property in the abstract system can be reproduced in the concrete system. Counterexamples corresponding to these behaviors are said to be *spurious*. When such a counterexample is found, the abstraction can be refined in order to eliminate the spurious behavior. This

[⋆] This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, the U.S. government or any other entity.

process is repeated until either a real counterexample is found, or the abstract system satisfies the property. In the latter case, we know that the concrete system satisfies the property as well, since the abstraction is conservative.

There are many known techniques, some automatic and some manual, for generating the initial abstraction and for abstraction/refinement. The automatic techniques are more relevant to this paper, not only because our method is fully automatic, but also because of the clear practical advantage of automation. Our methodology is based on an iterative abstraction/refinement process. Abstraction is performed by selecting a set of latches or variables and making them *invisible*, i.e., they are treated as inputs. In each iteration, we check whether the abstract system satisfies the specification with a standard OBDD-based symbolic model checker. If a counterexample is reported by the model checker, we try to simulate it on the concrete system with a fast SAT solver. In other words, we generate and solve a SAT instance that is satisfiable if and only if the counterexample is real. If the instance is not satisfiable, we look for the *failure state*, which is the last state in the longest prefix of the counterexample that is still satisfiable. Note that this process can not be easily performed with a standard circuit simulator, because the abstract counter example does not include values for all inputs.

We use the failure state in order to refine the abstraction. The abstract system has transitions from the failure state that do not exist in the concrete system. We eliminate these transitions by refining the abstraction, i.e., by making some variables visible that were previously invisible. The problem of selecting a small set of variables to make visible is one of the main issues that we address in this paper. It is important to find a small set in order to keep the size of the abstract state space manageable. This problem can be reduced to a problem of separating two sets of states (abstraction unites concrete states, and therefore refining an abstraction is the opposite operation, i.e., separation of states). For realistic systems, generating these sets is not feasible, both explicitly and symbolically. Moreover, the minimum separation problem is known to be NP-hard [5]. We combine *sampling* with Integer Linear Programming (ILP) and machine learning to handle this problem. Machine learning algorithms are successfully used in a wide range of problem domains like data mining and other problems where it is necessary to extract implicit information from a large database of samples[10]. These algorithms exploit ideas from a diverse set of disciplines, including information theory, statistics and complexity theory.

The closest work to the current one that we are aware of was described in [5]. Like the current work, they also use an automatic, iterative abstraction/refinement procedure that is guided by the counterexample, and they also try to eliminate the counterexample by solving the state-separation problem. But there are three main differences between the two methods. First, their abstraction is based on replacing predicates of the program with new input variables, while our abstraction is performed by making some of the variables invisible (thus, we hide the entire logic that defines these variables). As we will later show, the advantage of this approach is that computing a minimal abstraction function becomes easy. Secondly, checking whether the counterexample is real or spurious was performed in their work symbolically, using OBDDs. We do this stage with a SAT solver, which for this particular task is extremely efficient (due to the large number of solutions to the SAT instance). Thirdly, they derive the refinement symbolically. Since finding the coarsest refinement is NP-hard, they present a polynomial procedure that in general computes a sub-optimal solution. For some well defined cases the same procedure computes the optimal refinement. We, on the other hand, avoid the complexity by considering only samples of the states sets, which we compute explicitly. By doing so we also pay the price of optimality: this procedure yields a refinement step which is not necessarily optimal (i.e., we do not necessarily find the smallest number of invisible variables that should become visible in order to eliminate the counterexample). Yet we suggest a method for efficient sampling, which in most cases allows us to efficiently compute an optimal refinement.

The work of [7] should also be mentioned in this context, since it is very similar to [5], the main difference being the refinement algorithm: rather than computing the refinement by analyzing the abstract failure state, they combine a theorem prover with a greedy algorithm that finds a small set of previously abstracted predicates that eliminate the counterexample. They add this set of predicates as a new constraint to the abstract model.

Previous work on abstraction by making variables invisible (this technique was used under different names in the past) include the localization reduction of Kurshan [8] and many others (see, for example [1, 9]). The localization reduction follows the typical abstraction/refinement iterative process. It starts by making all but the property variables invisible. When a spurious counterexample is identified, it refines the system by making more variables visible. The variables made visible are selected according to the variable dependency graph and information that is derived from the counterexample. The candidates in the next refinement step are those invisible variables that are adjacent on the variable dependency graph to currently visible variables. Choosing among these variables is done by extracting information from the counterexample. Another relevant work is described in [15]. They use 3-valued simulation to simulate the counterexample on the concrete model and identify the invisible variables whose values in the concrete model conflict with the counterexample. Variables are chosen from this set of invisible variables by various ranking heuristics. For example, like localization, they prefer variables that are close on the variable dependency graph to the currently visible variables.

The rest of the paper is organized as follows. In the next section we briefly give the technical background of abstraction and refinement in model checking. In section 3 we describe our counterexample guided abstraction/refinement framework. We elaborate in this section on how the counterexample is being checked and how we refine the abstraction. We also describe refinement as a learning problem. In sections 4 and 5 we elaborate on our separation techniques. These techniques are combined with the efficient sampling technique, which is described in section 6. We give experimental results in section 7, which proves the viability of our methods comparing to a state of the art model checker (Cadence SMV). We discuss conclusions and future work in section 8.

2 Abstraction in Model Checking

We start with a brief description of the use of abstraction in model checking (for more details refer to [1]). Consider a program with a set of variables $V = \{x_1, \ldots, x_n\}$, where each variable x_i ranges over a non-empty domain D_{x_i}. Each state s of the program assigns values to the variables in V. The set of all possible states for the program is $S = D_{x_1} \times \cdots \times D_{x_n}$. The program is modeled by a transition system $M = (S, I, R)$ where

1. S is the set of states.
2. $I \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is the set of transitions.

We use the notation $I(s)$ to denote the fact that a state s is in I, and we write $R(s_1, s_2)$ if the transition between the states s_1 and s_2 is in R.

An abstraction function h for the system is given by a surjection $h : S \to \hat{S}$, which maps a concrete state in S to an abstract state in $\hat{S}$. Given a concrete state $s_i \in S$, we denote by $h(s_i)$ the abstract state to which it is mapped by h. Accordingly, we denote by $h^{-1}(\hat{s})$ the set of states s such that $h(s) = \hat{s}$.

Definition 1. *The* minimal abstract transition system $\hat{M} = (\hat{S}, \hat{I}, \hat{R})$ *corresponding to a transition system* $M = (S, I, R)$ *and an abstraction function* h *is defined as follows:*

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

Intuitively, minimality means that $\hat{M}$ can start in state $h(s)$ if and only if M can start in state s , and $\hat{M}$ can transition from $h(s)$ to $h(s')$ if and only if M can transition from s to s'.

For simplicity, we restrict our discussion to model checking of $\mathbf{AG}p$ formulas, where p is a non-temporal propositional formula. The theory can be extended to handle any safety property, because such formulas have counterexamples that are finite paths.

Definition 2. *A propositional formula* p respects *an abstraction function* h *if for all* $s \in S$, $h(s) \models p \Rightarrow s \models p$.

The essence of conservative abstraction is the following preservation theorem[6], which is stated without proof.

Theorem 1. *Let* $\hat{M}$ *be an abstraction of* M *corresponding to the abstraction function* h, *and* p *be a propositional formula that respects* h. *Then* $\hat{M} \models \mathbf{AG}p \Rightarrow M \models \mathbf{AG}p$

The converse of the above theorem is not true, however. Even if the abstract model invalidates the specification, the concrete model may still satisfy the specification. In this case, the abstract counterexample generated by the model checker is *spurious*, i.e. it does not correspond to a concrete path. The abstraction function is too coarse to validate the specification, and we need to refine it.

Definition 3. *Given a transition system $M = (S, I, R)$ and an abstraction function h, h' is a* refinement *of h if*

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

3 Abstraction-Refinement

Based on the above definitions, we now describe our *counterexample guided abstraction refinement* procedure. Given a transition system M and a safety property φ:

1. Generate an initial abstraction function h.
2. Model check $\hat{M}$. If $\hat{M} \models \varphi$, then $M \models \varphi$. Return TRUE.
3. If $\hat{M} \not\models \varphi$, check the counterexample on the concrete model. If the counterexample is real, $M \not\models \varphi$. Return FALSE.
4. Refine h, and go to step 2.

The above procedure is complete for finite state systems. Since each refinement step partitions at least one abstract state, the number of loop iterations is bounded by the number of concrete states. In the next subsections, we explain in more detail how we perform each step.

3.1 Defining an abstraction function

We partition the set of variables V into two sets: the set of *visible* variables which we denote by $\mathcal{V}$ and the set of *invisible* variables which we denote by $\mathcal{I}$. Intuitively, $\mathcal{V}$ corresponds to the part of the system that is currently believed to be important for verifying the property. The abstraction function h abstracts out the irrelevant details, namely the invisible variables. The initial abstraction in step 1 and the refinement in step 4 correspond to different partitions of the set of variables. As an initial abstraction, $\mathcal{V}$ includes the variables in the property φ. In each refinement step, we move variables from $\mathcal{I}$ to $\mathcal{V}$, as we will explain in sub-section 3.3.

More formally, let $s(x)$, $x \in V$ denote the value of variable x in a state s. Given a set of variables $U = \{u_1, \ldots, u_p\}$, $U \subseteq V$, s^U denotes the portion of s that corresponds to the variables in U, *i.e.* $s^U = (s(u_1) ... s(u_p))$. Let $\mathcal{V} = \{v_1, \ldots, v_k\}$. The partitioning defines our abstraction function $h : S \to \hat{S}$: The set of abstract states is $\hat{S} = D_{v_1} \times \cdots \times D_{v_k}$ and the abstraction function is simply $h(s) = s^{\mathcal{V}}$.

Given h, we need to compute the minimal abstraction. For an arbitrary system M and abstraction function h, it is often too expensive or impossible to construct the minimal abstraction $\hat{M}$[6]. However, our abstraction function allows us to compute $\hat{M}$ efficiently for systems where the transition relation R is in a functional form, e.g. sequential circuits. For these systems, $\hat{M}$ can be computed directly from the program text, by removing the logic that defines the invisible variables and treating them as inputs.

3.2 Checking the Counterexample

For safety properties, the counterexample generated by the model checker is a path $\langle \hat{s}_1, \hat{s}_2, \ldots \hat{s}_m \rangle$. The set of concrete paths that corresponds to this counterexample is given by

$$\psi_m = \{\langle s_1 \ldots s_m \rangle \mid I(s_1) \wedge \bigwedge_{i=1}^{m-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=1}^{m} h(s_i) = \hat{s}_i\} \tag{1}$$

According to section 3.1, $h(s_i)$ is simply a projection of s_i to the visible variables. The right-most conjunct is therefore a restriction of the visible variables in step i to their values in the counterexample.

The counterexample is spurious if and only if the set ψ_m is empty. We check for that by solving ψ_m with a SAT solver. This formula is very similar in structure to the formulas that arise in Bounded Model Checking(BMC)[3]. However, ψ_m is easier to solve because the path is restricted to the counterexample. Most model checkers treat inputs as latches, and therefore the counterexample includes assignments to inputs. While simulating the counterexample, we also restrict the values of the (original) inputs that are part of the definition (lie on the RHS) of the visible variables, which further simplifies the formula.

If a satisfying assignment is found, we know that the counterexample corresponds to a concrete path, which means that we found a real bug. Otherwise, we try to look for the 'failure' index f, i.e. the maximal index f, $f < m$, such that ψ_f is satisfiable. Given f, $\langle \hat{s}_1, \ldots \hat{s}_f \rangle$ is the longest prefix of the counterexample that corresponds to a concrete path. Our implementation sequentially searches in the range $1..m$ for the highest value f such that ψ_f is satisfiable. For long counterexample traces, we also have an option of performing a binary search over this range, in which case the number of SAT instances we solve is bounded by $\log m$.

3.3 Refining the abstraction

As before, let f denote the failure index. Let D denote the set of all states d_f such that there exists some $\langle d_1 ... d_f \rangle$ in ψ_f. We call D the set of *deadend* states. By definition, there is no concrete transition from D to $h^{-1}(\hat{s}_{f+1})$.

Since there is an abstract transition from $\hat{s}_f$ to $\hat{s}_{f+1}$, there is a non-empty set of transitions ϕ_f from $h^{-1}(\hat{s}_f)$ to $h^{-1}(\hat{s}_{f+1})$ that agree with the counterexample.

The set of transitions ϕ_f is defined as follows:

$$\phi_f = \{\langle s_f, s_{f+1}\rangle \mid R(s_f, s_{f+1}) \wedge h(s_f) = \hat{s}_f \ \wedge h(s_{f+1}) = \hat{s}_{f+1}\} \tag{2}$$

Given the definition of h, ϕ_f represents all concrete paths from step f to step $f+1$, where the visible variables in these steps are restricted to their values in the counterexample. Let B denote the set of all states b_f such that there exists some $\langle b_f, b_{f+1}\rangle$ in ϕ_f. We call B the set of *bad* states (see figure 1).

The counterexample exists because there is an abstract transition from s_f to s_{f+1} that does not correspond to any concrete transition. The transition exists because the deadend and bad states lie in the same abstract state. This suggests a mechanism to refine the abstraction. The abstraction h is refined to a new abstraction h' such that $\forall d \in D, \forall b \in B\ (h'(d) \neq h'(b))$. The new abstraction puts the deadend and bad states into separate abstract states and therefore eliminates the spurious transition from the abstract system.

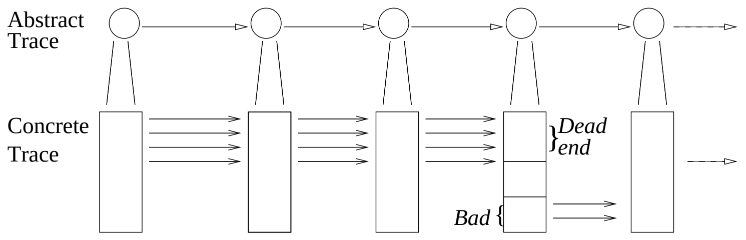


Fig. 1. A spurious counterexample corresponds to a concrete path that 'breaks' in the failing state. The failing state unites concrete 'deadend' and 'bad' states

3.4 Refinement by Separation and Learning

Let $S = \{s_1 ... s_m\}$ and $T = \{t_1 ... t_n\}$ be two sets of states (binary vectors) of size l, representing assignments to a set of variables W.

Definition 4. (The state separation problem) *Find a minimal set of variables $U = \{u_1 ... u_k\}$, $U \subset W$, such that for each pair of states (s_i, t_j), $1 \leq i \leq m$, $1 \leq j \leq n$, there exists a variable $u_r \in U$ such that $s_i(u_r) \neq t_j(u_r)$.*

Let D_I and B_I denote the restriction of D and B, respectively, to their invisible parts, *i.e.*, $D_I = \{s^I | s \in D\}$ and $B_I = \{s^I | s \in B\}$. Let $H \in \mathcal{I}$ be a set of variables that separates D_I from B_I. The refinement is obtained by adding H to $\mathcal{V}$. Minimality of H is not crucial, rather it is a matter of efficiency. Smaller sets of visible variables make it easier to model check the abstract system, but can also be harder to find. In fact, it can be shown that computing the minimal separating set is NP-hard[5].

Lemma 1. *The new abstraction function h' separated D from B in the abstract system.*

Proof. Let $d \in D$ and $b \in B$. The refined abstraction function h' corresponds to the visible set $\mathcal{V}' = \mathcal{V} \cup H$. Since H separates D_I and B_I, there exists a $u \in H$ s.t. $d(u) \neq b(u)$. Thus, for some $u \in \mathcal{V}'$, $d(u) \neq b(u)$. By definition, $h'(d) = (d(u_1)...d(u_k))$ and $h'(b) = (b(u_1)...b(u_k))$, $u_i \in \mathcal{V}'$. Thus, $h'(d) \neq h'(b)$. □

The naive way of separating the set of deadend states D from the set of bad states B would be to generate and separate D and B, either explicitly or symbolically. Unfortunately, for systems of realistic size, this is usually not possible. For all but the simplest examples, the number of states in D and B is too large to enumerate explicitly. For systems with moderate complexity, these sets can be computed symbolically with BDDs. However, even this is not possible for larger systems. Moreover, even if it were possible to generate D and B, it would still be computationally expensive to identify the separating variables.

Instead, we select *samples* from D and B and try to infer the separating variables for the entire sets from these samples. Of course, there is a tradeoff between the computational complexity of generating the samples, and the quality of the separating variables. Without a complete separation of D and B it can not be guaranteed that the counterexample will be eliminated. However, our algorithm is complete, because the counterexample will eventually be eliminated in subsequent refinement iterations. Our experience shows that state of the art SAT solvers like Chaff[11] can generate many samples in a short amount of time. The fact that D and B are large makes it relatively easy for SAT solvers to find satisfying assignments to equations 1 and 2 compared to typical SAT instances of similar size.

The idea of learning from samples has been studied extensively in the machine learning literature. A number of learning models and algorithms have been proposed. In the next two sections, we describe the techniques that we used to separate sets of samples of deadend and bad states, denoted by S_{D_I} and S_{B_I} respectively.

4 Separation as an Integer Linear Programming problem

A formulation of the problem of separating S_{D_I} from S_{B_I} as an Integer Linear Programming (ILP) problem is depicted in Figure 2.

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

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

Fig. 2. State Separation with Integer Linear Programming

The value of each integer variable[1] $v_1 ... v_{|\mathcal{I}|}$ in the ILP problem is interpreted as: $v_i = 1$ if and only if v_i is in the separating set. Every constraint corresponds to a pair of states (s_i, t_j), stating that at least one of the variables that separates (distinguishes) between the two states should be selected. Thus, there are $|S_{D_I}| \times |S_{B_I}|$ constraints.

Example 1. Consider the following two pairs of states: $s_1 = (0,1,0,1), s_2 = (1,1,1,0)$ and $t_1 = (1,1,1,1), t_2 = (0,0,0,1)$. The corresponding ILP problem will be

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

subject to:

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

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

5 Separation using Decision Tree Learning

The ILP-based separation algorithm outputs the minimal separating set. However, the algorithm has a high complexity and cannot handle a large number of variables or samples. In this section, we formulate the separation problem as a Decision Tree Learning(DTL) problem, which is polynomial both in the number of variables and the number of samples.

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

Data is classified by starting at the root node of the decision tree, testing the attribute specified by this node, and then moving down the tree branch corresponding to the value of the attribute. The process is repeated for the subtree rooted at the branch until one of the leafs is reached, which is labeled with the classification. The problem of separating S_{D_I} from S_{B_I} can be formulated as a DTL problem as follows:

– The attributes correspond to the invisible variables.

[1] Although the ILP problem is stated for integer variables, the constraints and objective function guarantees that their value will be either 0 or 1. Thus, they can be thought of as Boolean variables.

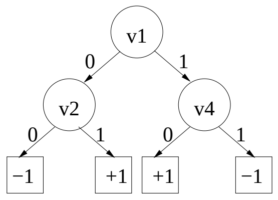
– The classifications are $+1$ and -1, corresponding to S_{D_I} and S_{B_I}.
– The examples are S_{D_I} labeled $+1$, and S_{B_I} labeled -1.

We generate a decision tree for this problem. The separating set that we output contains all the variables present at an internal nodes of the decision tree.

Lemma 2. *The above algorithm outputs a separating set for S_{D_I} and S_{B_I}.*

Proof. Let $d \in S_{D_I}$ and $b \in S_{B_I}$. The decision tree will classify d as $+1$ and b as -1. So, there exists a node n in the decision tree, labeled with a variable v, such that $d(v) \neq b(v)$. By construction, v lies in the output set. □

Example 2. Going back to example 1, the corresponding DTL problem has 4 attributes (v_1, v_2, v_3, v_4) and as always, two classifications $(+1, -1)$. The set of examples is $E = \{((0,1,0,1),+1), ((1,1,1,0),+1), ((1,1,1,1),-1), ((0,0,0,1),-1)\}$. The following tree corresponds to the separating set (v_1, v_2, v_4).



A number of algorithms have been developed for learning decision trees, e.g. ID3[12], C4.5[13]. All these algorithms essentially perform a simple top-down greedy search through the space of possible decision trees. We implemented a simplified version of the ID3 algorithm, which is described in Figure 3[10]. At

```
$DecTree(Examples, Attributes)$

 1. Create a $Root$ node for the tree.
 2. If all examples are classified the same, return $Root$ with this classification.
 3. Let $A = BestAttribute(Examples, Attributes)$. Label $Root$ with attribute $A$.
 4. For $i \in \{0,1\}$, let $Examples_i$ be the subset of $Examples$ having value $i$ for $A$.
 5. For $i \in \{0,1\}$, add an $i$ branch to the $Root$ pointing to subtree generated by
    $Dectree(Examples_i, Attributes - \{A\})$.
 6. return $Root$.
```

Fig. 3. Decision Tree Learning Algorithm

each recursion, the algorithm has to pick an attribute to test at the root. We need a measure of the quality of an attribute. We start with defining a quantity called *entropy*, which is a commonly used notion in information theory. Given a set S containing $n_\oplus$ positive examples and $n_\ominus$ negative examples, the entropy of S is given by:

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

where $p_\oplus = (n_\oplus)/(n_\oplus + n_\ominus)$ and $p_\ominus = (n_\ominus)/(n_\oplus + n_\ominus)$. Intuitively, entropy characterizes the variety in a set of examples. The maximum value for entropy is 1, which corresponds to a collection that has an equal number of positive and negative examples. The minimum value of entropy is 0, which corresponds to a collection with only positive or only negative examples. We can now define the quality of an attribute A by the reduction in entropy on partitioning the examples using A. This measure, called the *information gain* is defined as follows:

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

where E_0 and E_1 are the subsets of examples having the value 0 and 1, respectively, for attribute A. The $BestAttribute(Examples, Attributes)$ procedure returns the attribute $A \in Attributes$ that has the highest $Gain(Examples, A)$.

Example 3. We illustrate the working of our algorithm with an example. Continuing with our previous example, we calculate the gains for the attributes at the top node of the decision tree.

$$\begin{aligned}
&Entropy(E) = -(2/4)log_2(2/4) - (2/4)log_2(2/4) = 1.00\\
&Gain(E, v_1) = 1-(2/4)\cdot Entropy(E_{v_1=0})-(2/4)\cdot Entropy(_{v_1=1}) = 0.00\\
&Gain(E, v_2) = 1-(1/4)\cdot Entropy(E_{v_2=0})-(3/4)\cdot Entropy(_{v_2=1}) = 0.31\\
&Gain(E, v_3) = 1-(2/4)\cdot Entropy(E_{v_3=0})-(2/4)\cdot Entropy(_{v_3=1}) = 0.00\\
&Gain(E, v_4) = 1-(1/4)\cdot Entropy(E_{v_4=0})-(3/4)\cdot Entropy(_{v_4=1}) = 0.31
\end{aligned}$$

The $DecTree$ algorithm will pick v_2 or v_4 to label the $Root$.

6 Efficient sampling of states

Sampling D_I and B_I does not have to be arbitrary. As we now show, it is possible to direct the search for samples that contain more information than others. Let $\delta(D_I, B_I)$ denote the minimal separating set for D_I and B_I. Finding $\delta(D_I, B_I)$ by explicitly computing D_I and B_I and separating them is too computationally expensive, because both the size of these sets and the optimal separation techniques are worst-case exponential. We therefore look for samples S_{D_I} and S_{B_I} that are small enough to compute and separate, and, on the other hand, maintain $\delta(S_{D_I}, S_{B_I}) = \delta(D_I, B_I)$. Finding these sets is what we refer to as efficient sampling.

We suggest an iterative algorithm for efficient sampling. Let $SepSet$ denote the current separating set. Initially, $SepSet = \emptyset$. In each step $i > 0$ the algorithm finds samples that are not separable by $SepSet$ that was computed in the previous iteration. Computing a new pair of dead-end and bad states that are not separable by $SepSet$, can be done by solving $\Phi(SepSet)$, as defined below:

$$\Phi(SepSet) \doteq \psi_f \wedge \phi'_f \wedge \bigwedge_{v_i \in SepSet} v_i = v'_i \tag{3}$$

where ψ_f and ϕ_f are the formulas representing the deadend and bad states as defined in equations 1 and 2. The prime symbol over ϕ_f denotes the fact that we replace each variable $v \in \phi_f$ with a new variable v' (note that otherwise, by definition, the conjunction of ψ_f with ϕ_f is unsatisfiable). The right-most clause in the above formula guarantees that the new samples of deadend and bad states are not separable by the current separating set.

Algorithm *Sample-and-Separate*, described in Figure 6, uses formula 3 to compute the minimal separating set of D_I and B_I without explicitly computing or separating them. In each step i, it finds samples $d_i \in D_I$ and $b_i \in B_I$ that are not separable by the current separating set *SepSet*. It then re-computes *SepSet* for the union of sets that were computed up to the current iteration. By repeating this process until no such samples exist, it guarantees that the resulting separating set separates D_I from B_I. Note that the size of *SepSet* can either increase or stay unchanged in each iteration.

```
$SepSet = \emptyset$;
$i = 0$;
repeat forever {
    If $\Phi(SepSet)$ is sat., derive $d_i$ and $b_i$ from solution; else exit;
    $SepSet = \delta(\bigcup_{j=0}^{i}\{d_j\},\ \bigcup_{j=0}^{i}\{b_j\})$;
    $i = i + 1$; }
```

Fig. 4. Algorithm *Sample-and-Separate* implements efficient sampling by iteratively searching for states that are not separable by the current separating set.

The algorithm in Figure 6 finds a single solution to $\Phi(SepSet)$ and hence a single pair of states d_i and b_i. However, the size of each sample can be larger. Larger samples may reduce the number of iterations, but also require more time to derive and separate. The optimal number of new samples in each iteration depends on various factors, like the efficiency of the SAT solver, the separation technique and the examined model. Our implementation lets the user control this process by adjusting two parameters: the number of samples generated in each iteration, and the maximum number of iterations.

7 Experimental Results

We implemented our framework inside NuSMV[4]. We use NuSMV as a front-end, for parsing SMV files and for generating abstractions. However, for actual model checking, we use Cadence SMV, which implements techniques like cone-of-influence reduction, cut-points, etc. We implemented a variant of the ID3[12] algorithm to generate decision trees. We use a public domain LP solver[2] to solve our integer linear programs. We use Chaff[11] as our SAT solver. Some modifications were made to Chaff to efficiently generate multiple state samples in a single run. Our experiments were performed on the "IU" family of circuits, which are various abstractions of an interface control circuit from Synopsys. All experiments were performed on a 1.5GHz Dual Athlon machine with 3Gb RAM and running Linux. No pre-computed variable ordering files were used in the experiments.

Circuit	SMV		Sampling - ILP				Sampling - DTL				Eff. Samp. - DTL			
	Time	BDD	Time	BDD	S	L	Time	BDD	S	L	Time	BDD	S	L
$IU30$	0.7	116909	0.1	1731	0	1	0.1	1731	0	1	**0.1**	1731	0	1
$IU35$	0.6	149496	0.1	2357	0	1	0.1	2357	0	1	**0.1**	2357	0	1
$IU40$	1.2	225544	6.3	21249	3	4	0.9	18830	5	6	**0.6**	11028	2	3
$IU45$	37.5	2554520	6.1	17702	3	4	1.1	18847	5	6	**0.7**	10634	2	3
$IU50$	23.3	2094723	19.7	100647	13	14	**9.8**	90691	13	14	24.0	1274240	4	17
$IU55$	-	-	-	-	-	-	2072	51703825	6	9	**3.0**	64386	1	6
$IU60$	-	-	7.8	183811	4	7	7.8	183811	4	7	**4.5**	109393	1	6
$IU65$	-	-	7.9	192806	4	7	7.9	192806	4	7	**3.8**	47546	1	5
$IU70$	-	-	8.1	192806	4	7	8.2	192806	4	7	**3.8**	47546	1	5
$IU75$	102.9	7068752	32.0	142546	9	10	24.5	397620	13	14	**24.1**	550872	2	7
$IU80$	603.7	39989682	31.7	215404	9	10	44.0	341018	13	14	**24.1**	186662	2	7
$IU85$	2832	76232788	33.1	230979	9	10	44.6	443785	13	14	**25.2**	198359	2	7
$IU90$	-	-	33.0	230979	9	10	44.6	443785	13	14	**25.4**	198359	2	7

Fig. 5. Model checking results for property 1.

The results are presented in Figure 5 and Figure 6. The two tables correspond to two different properties. We compared the following techniques: 1) 'SMV': Cadence SMV, 2) 'Sampling-ILP': Sampling, separation using Integer Linear Programming, 50 samples per refinement iteration, 3) 'Sampling-DTL': Sampling, separation using Decision Tree Learning, 50 samples per refinement iteration, 4) 'Eff. Samp.-DTL': Efficient sampling, separation using Decision Tree Learning. For each run, we measured the total running time ('Time'), the maximum number of BDD nodes allocated ('BDD'), the number of refinement steps ('S'), and the number of latches in the final abstraction ('L'). The original number of latches in each circuit in indicated in its name. A '$-'$ symbol indicates that we ran out of memory. We could not solve Property 2 for circuits $IU55 ... IU70$ with any of the methods.

The experiments indicate that our technique expedites standard model checking, both in terms of execution time and required memory. As predicted, the number of iterations is generally reduced when either ILP or efficient sampling is applied. In most cases, this translates to a reduction in the total execution time. There were cases, however, when smaller sets of separating variables resulted in larger BDDs. Such 'noise' in the experimental results is typical of BDD based techniques.

8 Conclusions and Future Work

We have presented an automatic counterexample guided abstraction-refinement algorithm that uses SAT, ILP and techniques from machine learning. Our algorithm outperforms standard model checking, both in terms of execution time and memory requirements. Our refinement technique is very general and can be extended to a large variety of systems. For example, in conjunction with predicate abstraction, we can apply our techniques to software model checking. There

Circuit	SMV		Sampling - ILP				Sampling - DTL				Eff. Samp. - DTL			
	Time	BDD	Time	BDD	S	L	Time	BDD	S	L	Time	BDD	S	L
$IU30$	7.3	324268	8.0	113189	3	20	7.5	113189	3	20	**6.5**	113189	3	20
$IU35$	19.1	679224	11.8	186097	4	21	12.7	186097	4	21	**11.0**	186097	4	21
$IU40$	53.6	1100956	25.9	260299	6	23	19.0	207199	5	22	**16.1**	207199	5	22
$IU45$	226.1	6060256	28.3	411952	5	22	25.3	411952	5	22	**22.1**	411952	5	22
$IU50$	1754	25102082	160.4	2046981	13	32	**85.1**	605501	10	27	15120	3791826	7	31
$IU75$	-	-	1080	3716255	21	38	586.7	1178039	16	33	**130.5**	1050007	5	26
$IU80$	-	-	1136	3378860	21	38	552.5	1158076	16	33	**153.4**	1009030	5	26
$IU85$	-	-	1162	3493143	21	38	581.2	1272915	16	33	**167.7**	1079043	5	26
$IU90$	-	-	965	3712477	20	37	583.3	1271915	16	33	**167.1**	1079043	5	26

Fig. 6. Model checking results for property 2.

are several future research directions to our work. We are currently exploring criteria other than the size of the separating set for characterizing a good refinement. We also want to explore other machine learning techniques to solve the state separation problem.

References

1. F. Balarin and A. Sangiovanni-Vinventelli. An iterative approach to language containment. *(CAV'94)*.
2. M. Berkelaar. lpsolve, version 2.0. Eindhoven Univ. Tech., The Netherlands.
3. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *(TACAS'99)*.
4. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Int. Journal of Software Tools for Technology Transfer*, 1998.
5. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *(CAV'00)*.
6. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. Prog. Lang. Sys.*, 16(5):1512–1542, 1994.
7. S. Das and D.L. Dill. Successive approximation of abstract transition relations. *(LICS'01)*.
8. R. Kurshan. *Computer aided verification of coordinating processes.* Princeton University Press, 1994.
9. J. Lind-Nielsen and H. Andersen. Stepwise CTL model checking of state/event systems. *(CAV'99)*.
10. T.M. Mitchell. *Machine Learning.* WCB/McGraw-Hill, 1997.
11. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *(DAC'01)*.
12. J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1986.
13. J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.
14. J.P.M. Silva and K.A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical Report TR-CSE-292996, University of Michigen, 1996.
15. D. Wang, P. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. *DAC'01.*

Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis[⋆]

Pankaj Chauhan[1] Edmund Clarke[1] James Kukula[3]
Samir Sapra[1] Helmut Veith[2] Dong Wang[1]

[1] Carnegie Mellon University [2] TU Vienna, Austria
[3] Synopsys Inc., Beaverton, OR

Abstract. We introduce a SAT based automatic abstraction refinement framework for model checking systems with several thousand state variables in the cone of influence of the specification. The abstract model is constructed by designating a large number of state variables as *invisible*. In contrast to previous work where invisible variables were treated as free inputs we describe a computationally more advantageous approach in which the abstract transition relation is approximated by *pre-quantifying* invisible variables during image computation. The abstract counterexamples obtained from model-checking the abstract model are symbolically simulated on the concrete system using a state-of-the-art SAT checker. If no concrete counterexample is found, a subset of the invisible variables is reintroduced into the system and the process is repeated. The main contribution of this paper are two new algorithms for identifying the relevant variables to be reintroduced. These algorithms monitor the SAT checking phase in order to analyze the impact of individual variables. Our method is complete for safety properties in the sense that – performance permitting – a property is either verified or disproved by a concrete counterexample. Experimental results are given to demonstrate the power of our method on real-world designs.

1 Introduction

Symbolic model checking has been successful at automatically verifying temporal specifications on small to medium sized designs. However, the inability of BDD based model checking to handle large state spaces of "real world" designs hinders the wide scale acceptance of these techniques. There have been advances on various fronts to push the limits of automatic verification. On the one hand, improving BDD based algorithms improves the ability to handle large state machines, while on the other hand, various abstraction algorithms reduce the size of the design by focusing only on relevant portions of the design. It is important to make improvements on both fronts for successful verification.

[⋆] This research is sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the Gigascale Silicon Research Center (GSRC), the National Science Foundation (NSF) under Grant No. CCR-9803774, and the Max Kade Foundation. One of the authors is also supported by Austrian Science Fund Project N Z29-INF. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, GSRC, NSF, or the United States Government.

A conservative abstraction is one which preserves all behaviors of a concrete system. Conservative abstractions benefit from a *preservation* theorem which states that the correctness of any universal (e.g. ACTL*) formulae 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. To get rid of a spurious counterexample, the abstraction needs to be made more precise via refinement. It is obviously desirable to automate this procedure.

This paper focuses on automating the abstraction process for handling large designs containing up to a few thousand latches. This means that using any computation on concrete systems based on BDDs will be too expensive. Abstraction refinement [1, 6, 8, 11, 13, 17] is a general strategy for automatic abstraction. Abstraction refinement usually involves the following process.

1. **Generation of Initial Abstraction.** It is desirable to derive the initial abstraction automatically.
2. **Model checking of abstract system**. If this results in a conclusive answer for the abstract system, then the process is terminated. For example, in case of existential abstraction, a "yes" answer for an ACTL* property in this step means that the concrete system also satisfies the property, and we can stop. However, if the property is false on the abstract system, an abstract counterexample is generated.
3. **Checking whether the counterexample holds on the concrete system**. If the counterexample is valid, then we have actually found a bug. Otherwise, the counterexample is *spurious* and the abstraction needs to be refined. Usually, refinement of abstraction is based on the analysis of counterexample(s) generated.

Our abstraction function is based on hiding irrelevant parts of the circuit by make a set of variables *invisible*. This simple abstraction function yields an efficient way to generate minimal abstractions, a source of difficulty in previous approaches. We describe two techniques to produce abstract systems by removing invisible variables. The first is simply to make the invisible variables into input variables. This is shown to be a minimal abstraction. However, this leaves a large number of input variables in the abstract system and, consequently, BDD based model checking even on this abstract system becomes very difficult [19]. We propose an efficient method to pre-quantify these variables on the fly during image computation. The resulting abstract systems are usually small enough to be handled by standard BDD based model checkers. We use an enhanced version [3, 4] of NuSMV [5] for this. If a counterexample is produced for the abstract system, we try to simulate it on the concrete system symbolically using a fast SAT checker (Chaff [16, 21] in our case).

The refinement is done by identifying a small set of invisible variables to be made visible. We call these variables the *refinement variables*. Identification of refinement variables is the main focus of this paper. Our techniques for identifying important variables are based on analysis of effective *boolean constraint propagation (BCP)* and *conflicts* [16] during the SAT checking run of the counterexample simulation. Recently, propositional SAT checkers have demonstrated tremendous success on various classes of SAT formulas. The key to the effectiveness of SAT checkers like Chaff [16], GRASP [18] and SATO [20] is non-chronological backtracking, efficient conflict driven learning of conflict clauses, and improved decision heuristics.

SAT checkers have been successfully used for Bounded Model Checking (BMC) [2], where the design under consideration is unrolled and the property is symbolically verified using SAT procedures. BMC is effective for showing the presence of errors. However, BMC is not at all effective for showing that a specification is true unless the diameter of the state space is known. Moreover, BMC performance degrades when searching for deep counterexamples. Our technique can be used to show that a specification is true and is able to search for deeper concrete counterexamples because of the guidance derived from abstract counterexamples.

The efficiency of SAT procedures has made it possible to handle circuits with a few thousand of variables, much larger than any BDD based model checker is able to do at present. Our approach is similar to BMC, except that the propositional formula for simulation is constrained by assignments to visible variables. This formula is *unsatisfiable* for a spurious counterexample. We propose heuristic scores based on backtracking and conflict clause information, similar to VSIDS heuristics in Chaff, and conflict dependency analysis algorithm to extract the reason for unsatisfiability. Our techniques are able to identify those variables that are critical for unsatisfiability of the formula and are, therefore, prime candidates for refinement. The main strength of our approach is that we use the SAT procedure itself for refinement. We do not need to invoke multiple SAT instances or solve separation problems as in [8].

Thus the main contributions of our work are, (a) use of SAT for counterexample validation, (b) refinement procedures based on SAT conflict analysis, and, (c) a method to remove invisible variables from the abstract system for computational efficiency.

Outline of the Paper

The rest of the paper is organized as follows. In the next section, we describe related work. Section 3 briefly reviews how abstraction is used in model checking and introduces notation that is used in the following sections. In Section 4, we describe in detail, our abstraction technique and how we check an abstract counterexample on the concrete model. The most important part of the paper is Section 5, where we discuss our refinement algorithms based on scoring heuristics for variables and conflict dependency analysis. In section 6, we present experimental evidence to show the ability of our approach to handle large state systems. Finally, we conclude in Section 7 with directions for future research.

2 Related Work

Our work compares most closely to that presented in [6] and more recently [8]. There are three major differences between our work and [6]. First, their initial abstraction is based on predicate abstraction, where new set of program variables are generated representing various predicates. They symbolically generate and manipulate these abstractions with BDDs. Our abstraction is based on hiding certain parts of the circuit. This yields an easier way to generate abstractions. Secondly, the biggest bottleneck in their method is the use of BDD based image computations *on concrete systems* for validating counterexamples. We use symbolic simulation based on SAT accomplish this task, as in [8]. Finally, their refinement is based on splitting the variable domains. The problem of finding the coarsest refinement is shown to be NP-hard in [6]. Because our abstraction functions are simpler, we can identify refinement variables during the SAT checking phase. We do not need to solve any other problem for refinement.

We differ from [8] in three aspects. First, we propose to remove invisible variables from abstract systems on the fly by quantification. This reduces the complexity of BDD based model checking of abstract systems. Leaving a large number of input variables in the system makes it very difficult to model check even an abstract system [19]. Secondly, computation overhead for our separation heuristics is minimal. In their approach, refinement is done by separating *dead-end* and *bad* states (sets of concrete states contained in the failure state) with ILP solvers or machine learning. This requires enumerating *all* dead-end and bad states or producing samples of these states and separating them. The sampling scheme they propose requires calling multiple instances of the SAT checker. Experiments on large circuits have shown that efficiently generating these samples is a major bottleneck in their method. We avoid this step altogether and cheaply identify refinement variables from the analysis of a single SAT check that is already done. We do not claim any optimality on the number of variables, however, this is a small price to pay for efficiency. We have been able to handle a circuit with about 5000 variables in cone of influence of the specification, for which their method gets stuck in the sampling phase. Finally, we believe our method can identify a better set of invisible registers for refinement. Although [8] uses optimization algorithms to minimize the number of registers to refine, their algorithm relies on sampling to provide the candidate separation sets. When the size of the problem becomes large, there could be many possible separation sets. The quality of the separating set can not be judged by its size, instead a better selection criteria is required. Our method is based on SAT conflict analysis. The Boolean constraint propagation (BCP) algorithm in a SAT solver naturally limits the number of candidates that we will need to consider. We use conflict dependency analysis to reduce further the number of candidates for refinement.

The work of [10] focuses on algorithms to refine an approximate abstract transition relation. Given a spurious abstract transition, they combine a theorem prover with a greedy strategy to enumerate the part of the abstract transition that does not have corresponding concrete transitions. The identified bad transition is removed from the current abstract model for refinement. Their enumeration technique is potentially expensive. More importantly, they do not address the problem of how to refine abstract predicates.

Previous work on abstraction by making variables invisible includes the localization reduction of Kurshan [13] and other techniques (e.g. [1, 14]). Localization reduction begins with the set of variables in the property as visible variables. The set of variables adjacent to the present set of visible variables in the variable dependency graph are chosen as the candidates for refinement. Counterexamples are analyzed in order to choose variables among these candidates.

The work presented in [19] combines three different engines (BDD, ATPG and simulation) to handle large circuits using abstraction and refinement. The main difference between our method and that in [19] is the strategy for refinement. In [19], candidates for refinement are based on those invisible registers that get assigned in the abstract counterexample. In our approach, we intentionally throw away invisible registers in the abstract counterexample, and rely on our SAT conflict analysis to select the candidates. We believe there are two advantages to disallowing invisible registers in the abstract counterexample. First of all, generating an abstract counterexample is computationally expensive, when the number of invisible registers is large. In fact, for efficiency reasons, a BDD/ATPG hybrid engine is used in [19] to model check the abstract model. By quantifying the invisible variables early, we avoid this bottleneck. More importantly, in [19], 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 visible registers and hence a real counterexample can be found more cheaply.

3 Abstraction in Model Checking

We give a brief summary of the use of abstraction in model checking and introduce notation that we will use in the remainder of the paper (refer to [7] for a full treatment). A transition system is modeled by a tuple $M = (S, I, R, \mathcal{L}, L)$ where S is the set of states, $I \subseteq S$ is the set of initial states, R is the set of transitions, $\mathcal{L}$ is the set of atomic propositions that label each state in S with the labeling function $L : S \to 2^{L}$. The set I is also used as a predicate $I(s)$, meaning the state s is in I. Similarly, the transition relation R is also used as a predicate $R(s_1, s_2)$, meaning there exists a transition between states s_1 and s_2. Each program variable v_i ranges over its non-empty domain D_{v_i}. The state space of a program with a set of variables $V = \{v_1, v_2, \ldots, v_n\}$ is defined by the Cartesian product $D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$.

In existential abstraction [7] a surjection $h : S \to \hat{S}$ maps a concrete state $s_i \in S$ to an abstract state $\hat{s}_i = h(s_i) \in \hat{S}$. We denote the set of concrete states that map to an abstract state $\hat{s}_i$ by $h^{-1}(\hat{s}_i)$.

Definition 1. *The **minimal existential abstraction** $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{\mathcal{L}}, \hat{L})$ corresponding to a transition system $M = (S, I, R, \mathcal{L}, L)$ and an abstraction function h is defined by:*

1. $\hat{S} = \{\hat{s} | \exists s. s \in S \wedge h(s) = \hat{s}\}$.
2. $\hat{I} = \{\hat{s} | \exists s. I(s) \wedge h(s) = \hat{s}\}$.
3. $\hat{R} = \{(\hat{s}_1, \hat{s}_2) | \exists s_1. \exists s_2. R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2\}$.
4. $\hat{\mathcal{L}} = \mathcal{L}$.
5. $\hat{L}(\hat{s}) = \bigcup_{h(s)=\hat{s}} L(s)$.

Condition 3 can be stated equivalently as

$$\exists s_1, s_2 (R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2) \Leftrightarrow \hat{R}(\hat{s}_1, \hat{s}_2) \qquad (1)$$

An atomic formula f *respects* h if for all $s \in S$, $h(s) \models f \Rightarrow s \models f$. Labeling $\hat{L}(\hat{s})$ is *consistent*, if for all $s \in h^{-1}(\hat{s})$ it holds that $s \models \bigwedge_{f \in \hat{L}(\hat{s})} f$. The following theorem from [6, 15] is stated without proof.

Theorem 1. *Let h be an abstraction function and ϕ an $ACTL^*$ specification where the atomic sub-formulae respect h. Then the following holds: (i) For all $\hat{s} \in \hat{S}$, $\hat{L}(\hat{s})$ is consistent, and (ii) $\hat{M} \models \phi \Rightarrow M \models \phi$.*

This theorem is the core of all abstraction refinement frameworks. However, the converse may not hold, i.e., even if $\hat{M} \not\models \phi$, the concrete model M may still satisfy ϕ. In this case, the counterexample on $\hat{M}$ is said to be spurious, and we need to refine the abstraction function. Note that the theorem holds even if only the right implication holds in Equation 1. In other words, even if we add more transitions to the minimal transition relation $\hat{R}$, the validity of an ACTL* formula on $\hat{M}$ implies its validity on M.

Definition 2. *An abstraction function h' is a* ***refinement*** *for the abstraction function h and the transition system $M = (S, I, R, \mathcal{L}, L)$ if for all $s_1, s_2 \in S, h'(s_1) = h'(s_2)$ implies $h(s_1) = h(s_2)$. Moreover, h' is a* ***proper refinement*** *of h if there exist $s_1, s_2 \in S$ such that $h(s_1) = h(s_2)$ and $h'(s_1) \neq h'(s_2)$.*

In general, ACTL* formulae can have *tree-like* counterexamples [9]. In this paper, we focus only on safety properties, which have finite path counterexamples. It is possible to generalize our approach to full ACTL* as done in [9]. The following iterative abstraction refinement procedure for a system M and a safety formula ϕ follows immediately.

1. Generate an initial abstraction function h.
2. Model check $\hat{M}$. If $\hat{M} \models \phi$, return `TRUE`.
3. If $\hat{M} \not\models \phi$, check the generated counterexample $\widehat{T}$ on M. If the counterexample is real, return `FALSE`.
4. Refine h, and goto step 2.

Since each refinement step partitions at least one abstract state, the above procedure is complete for finite state systems for ACTL* formulae that have path counterexamples. Thus the number of iterations is bounded by the number of concrete states. However, as we will show in the next two sections, the number of refinement steps can be at most equal to the number of program variables.

We would like to emphasize that we model check abstract system in step 2 using BDD based symbolic model checking, while steps 3 and 4 are carried out with the help of SAT checkers.

4 Generating Abstract State Machine

We consider a special type of abstraction for our methodology, wherein, we hide a set of variables that we call *invisible* variables, denoted by $\mathcal{I}$. The set of variables that we retain in our abstract machine are called *visible* variables, denoted by $\mathcal{V}$. The visible variables are considered to be important for the property and hence are retained in the abstraction, while the invisible variables are considered irrelevant for the property. The initial abstraction and the refinement in steps 1 and 4 respectively correspond to different partitions of V. Typically, we would want $|\mathcal{V}| \ll |\mathcal{I}|$. Formally, the value of a variable $v \in V$ in state $s \in S$ is denoted by $s(v)$. Given a set of variables $U = \{u_1, u_2, \ldots, u_p\}, U \subseteq V$, let s^U denote the portion of s that corresponds to the variables in U, i.e., $s^U = (s(u_1)s(u_2)\ldots s(u_p))$. Let $\mathcal{V} = \{v_1, v_2, \ldots, v_k\}$. This partitioning of variables defines our abstraction function $h : S \to \hat{S}$. The set of abstract states is $\hat{S} = D_{v_1} \times D_{v_2} \ldots \times D_{v_k}$ and $h(s) = s^{\mathcal{V}}$.

In our approach, the initial abstraction is to take the set of variables mentioned in the property as visible variables. Another option is to make the variables in the cone of influence (COI) of the property visible. However, the COI of a property may be too large and we may end with a large number of visible variables. The idea is to begin with a small set of visible variables and then let the refinement procedure come up with a small set of invisible variables to make visible.

We also assume that the transition relation is described not as a single predicate, but as a conjunction of bit relations R_j of each individual variable v_j. More formally, we consider a sequential circuit with registers $V = \{v_1, v_2, \ldots, v_m\}$ and inputs $I = \{i_1, i_2, \ldots, i_q\}$. Let $s = (v_1, v_2, \ldots, v_m)$, $i = (i_1, i_2, \ldots, i_q)$ and $s' = (v'_1, v'_2, \ldots, v'_m)$. The primed variables denote the next state versions of unprimed variables as usual. Thus the bit relation for v_j becomes $R_j(s, i, v'_j) = (v'_j \leftrightarrow f_{v_j}(s, i))$.

$$R(s, s') = \exists i \bigwedge_{j=1}^{m} R_j(s, i, v'_j) \tag{2}$$

4.1 Abstraction by Making Invisible Variables as Input Variables

As shown in [8], the minimal transition relation $\hat{R}$ corresponding to R and h described above is obtained by removing the logic defining invisible variables and treating them as free input variables of the circuit. Hence, $\hat{R}$ looks like:

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}} \exists i \bigwedge_{v_j \in \mathcal{V}} R_j(s^{\mathcal{V}}, s^{\mathcal{I}}, i, v'_j) \tag{3}$$

The quantifications in Equation 3 are performed during each image computation in symbolic model checking of the abstract system. This is done so as not to build a monolithic BDD for $\hat{R}$ and enjoy the benefits of early quantification.

We call this type of abstraction an *input abstraction*. We write s as $s^{\mathcal{V}}, s^{\mathcal{I}}$ to stress the fact that we are leaving invisible variables as input variables in $\hat{R}$. When dealing with systems with a large number of registers, quantifying so many variables for each image computation is expensive (e.g. [19]). An invisible variable can in the support of multiple partitions of the transition relation. In input abstraction, each occurence of an invisible variable has the same value in different partitions of the abstract transition relation. Thus, we say input abstraction preserves *correlations* between different occurrences of an invisible variable. In the next type of abstraction, we pre-quantify most of the invisible variables, to reduce the number of variables during image computation. This means that different occurrences of an invisible variable get de-coupled when we push the quantifications inside Equation 3, making the abstraction more approximate.

4.2 Abstraction by Pre-quantifying Invisible Variables

Input abstraction leaves a large number of variables to quantify during the image computation process. We can however, quantify these variables a priori, leaving only visible variables in $\hat{R}$. The transition relation that we get by quantifying invisible variables from $\hat{R}$ in the beginning is denoted by $\tilde{R}$. We can even quantify some of the input variables a priori in this fashion to control the total number of variables appearing in $\tilde{R}$. Let $Q \subseteq \mathcal{I} \cup I$ denote the set of variables to be pre-quantified and let $W = (\mathcal{I} \cup I) \setminus Q$, the set of variable that are not pre-quantified.

Quantification of a large number of invisible variables in Equation 3 is computationally expensive [15]. To alleviate this difficulty, it is customary to approximate this abstraction by pushing the quantification inside conjunctions as follows.

$$\tilde{R}(\hat{s}, \hat{s}') = \exists s^W \bigwedge_{v_j \in \mathcal{V}} \exists s^Q R_j(s^{\mathcal{V}}, s^{\mathcal{I}}, i, v_j') \tag{4}$$

Since the BDDs for state sets do not contain input variables in the support, this is a safe step to do. This does not violate the soundness of the approximation, i.e., for each concrete transition in R, there will be a corresponding transition in $\hat{R}$, as stated below.

Theorem 2. $\exists s_1, s_2(R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2) \Rightarrow \tilde{R}(\hat{s}_1, \hat{s}_2)$.

The other direction of this implication does not hold because of the approximations introduced.

Preserving Correlations We can see in Equation 4 that by existentially quantifying each invisible variable separately for each conjunct of the transition relation, we lose the correlation between different occurrences of a variable. For example, consider the trivial bit relations $x_1' = x_3, x_2' = \neg x_3$ and $x_3 = x_1 \oplus x_2$. Suppose x_3 is made an invisible variable. Then quantifying x_3 from the bit relations of x_1 and x_2 will result in the transition relation being always evaluated 1, meaning the state graph is a clique. However, we can see that in any reachable state, x_1 and x_2 are always opposite of each other. To solve this problem partially without having to resort to equation 4, we propose to cluster those bit relations that share many common variables. Since this problem is very similar to the quantification scheduling problem (which occurs during image computations), we propose to use a modification of *VarScore* algorithms [3] for evaluating this quantification. This algorithm can be viewed as producing clusters of bit relations. We use it to produce clusters with controlled approximations. The idea is to delay variable quantifications as much as possible, without letting the conjoined BDDs grow too large. When a BDD grows larger than some threshold, we quantify away a variable. We can of course quantify a variable that no longer appears in the support of other BDDs. Effective quantification scheduling algorithms put closely related occurrences of a variable in the same cluster. Figure 1 shows the VarScore algorithm for approximating existential abstraction.

A static circuit minimum cut based structural method to reduce the number of invisible variables was proposed in [12] and used in [19]. Our method introduces approximations as needed based on actual image computation, while there method removes the variables statically. Our algorithms achieves a balance between performance and accuracy. This means that the approximations introduced by our algorithm are more accurate as the parts of the circuits statically removed in [12] could be important.

4.3 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 $\bar{s}_m = \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_m \rangle$. To check whether this counterexample holds on the concrete model M or not, we symbolically

```
Given a set of conjuncts $R_V$ and variables $s^Q$ to pre-quantify
Repeat until all $s^Q$ variables are quantified

1. Quantify away $s_Q$ variables appearing in only one BDD
2. Score the variables by summing up the sizes of BDDs in which a variable occurs
3. Pick two smallest BDDs for the variable with the smallest score
4. If any BDD is larger then the size threshold, quantify the variable from BDD(s) and
   go back to step 2.
5. If the BDDs are smaller than threshold, do BDDAnd or BDDAndExists depending
   upon the case
```

Fig. 1. VarScore algorithm for approximating existential abstraction

simulate M beginning with the initial state $I(s_0)$ using a fast SAT checker. At each stage of the symbolic simulation, we constrain the values of visible variables only according to the counterexample produced. The equation for symbolic simulation is:

$$\begin{aligned}(I(s_0) \wedge (h(s_0) = \hat{s}_0)) \wedge (R(s_0, s_1) \wedge (h(s_1) = \hat{s}_1)) \wedge \ldots \\ \wedge(R(s_{m-1}, s_m) \wedge (h(s_m) = \hat{s}_m))\end{aligned} \tag{5}$$

Each $h(s_i)$ is just a projection of the state s_i onto visible variables. If this propositional formula is satisfiable, then we can successfully simulate the counterexample on the concrete machine to conclude that $M \not\models \phi$. The satisfiable assignments to invisible variables along with assignments to visible variables produced by model checking give a valid counterexample on the concrete machine.

If this formula is not satisfiable, the counterexample is *spurious* and the abstraction needs refinement. Assume that the counterexample can be simulated up to the abstract state $\hat{s}_f$, but not up to $\hat{s}_{f+1}$ ([6, 8]). Thus formula 6 is satisfiable while formula 7 is *not* satisfiable, as shown in Figure 2.

$$\begin{aligned}(I(s_0) \wedge (h(s_0) = \hat{s}_0)) \wedge (R(s_0, s_1) \wedge (h(s_1) = \hat{s}_1)) \wedge \ldots \\ \wedge(R(s_{f-1}, s_f) \wedge (h(s_f) = \hat{s}_f))\end{aligned} \tag{6}$$

$$\begin{aligned}(I(s_0) \wedge (h(s_0) = \hat{s}_0)) \wedge (R(s_0, s_1) \wedge (h(s_1) = \hat{s}_1)) \wedge \ldots \\ \wedge(R(s_f, s_{f+1}) \wedge (h(s_{f+1}) = \hat{s}_{f+1}))\end{aligned} \tag{7}$$

Using the terminology introduced in [6], we call the abstract state $\hat{s}_f$ a *failure state*. The abstract state $\hat{s}_f$ contains many concrete states given by all possible combinations of invisible variables, keeping the same values for visible variables as given by $\hat{s}_f$. The concrete states in $\hat{s}_f$ reachable from the initial states following the spurious counterexample are called the *dead-end* states. The concrete states in $\hat{s}_f$ that have a reachable set in $\hat{s}_{f+1}$ are called *bad* states. Because the dead-end states and the bad states are part of the same abstract state, we get the spurious counterexample. The refinement step then is to separate dead-end states and bad states by making a small subset of invisible variables visible. It is easy to see that the set of dead-end states are given by the values of state variables in the f^{th} step for all satisfying solutions to Equation 6.

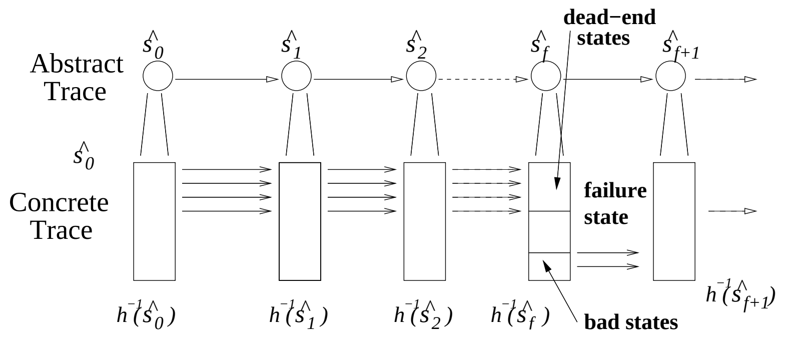


Fig. 2. A spurious counterexample showing failure state [8]. No concrete path can be extended beyond failure state.

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 [16]. We assume that there are concrete transitions which correspond to each abstract transition from $\hat{s}_i$ to $\hat{s}_{i+1}$, where $0 < i \leq f$. It is fairly straightforward to extend our algorithm to handle spurious abstract transitions. In this case, the set of *bad* states is not empty. Since $\bar{s}_f$ is the shortest prefix that is unsatisfiable, there must be information passed through the invisible registers at time frame f in order for the SAT solver to prove the counterexample is spurious. Specifically, the SAT solver implicitly generates constraints on the invisible registers at time frame f based on either the last abstract transition or the prefix $\bar{s}_f$. 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 f during the SAT process is sufficient to separate *deadend* states and *bad* states after refinement. Therefore, our algorithm limits the refinement candidates to the registers that are constrained in time frame f.

Equation 5 is exactly like symbolic simulation with Bounded Model Checking. The only difference is that the values of visible state variables at each step are constrained to the counterexample values. Since the original input variables to the system are unconstrained, we also constrain their values according to the abstract counterexample. This puts many constraints on the SAT formula. Hence, the SAT checker is able to prune the search space significantly. We rely on the ability of Chaff to identify important variables in this SAT check to separate dead-end and bad states, as described in the next section.

5 SAT Based Refinement Heuristics

The basic framework for these SAT procedures is Davis-Putnam-Logeman-Loveland backtracking search, shown in Figure 3. The function `decide_next_branch()` chooses the branching variable at current *decision level*. The function `deduce()` does *Boolean constraint propagation* to deduce further assignments. While doing so, it might infer that the present set of assignments to variables do not lead to any satisfying solution, leading to a conflict. In case of a conflict, new clauses are learned by `analyse_conflict()`

```
while(1) {
   if (decide_next_branch()) {        // Branching
      while (deduce() == conflict) {  // Propagate implications
         blevel = analyse_conflict(); // Learning
         if (blevel == 0)
            return UNSAT;
         else
            backtrack(blevel);        // Non-chronological
                                      // backtrack
      }
   }
   else                               // no branch means all vars
                                      // have been assigned
      return SAT;
}
```

Fig. 3. Basic DPLL backtracking search (used from [16] for illustration purpose)

that hopefully prevent the same unsuccessful search in the future. The conflict analysis also returns a variable for which another value should be tried. This variable may not be the most recent variable decided, leading to a *non-chronological* backtrack. If all variables have been decided, then we have found a satisfying assignment and the procedure returns. The strength of various SAT checkers lies in their implementation of constraint propagation, decision heuristics, and learning.

Modern SAT checkers work by introducing conflict clauses in the learning phase and by non-chronological backtracking. Implication graphs are used for Boolean constraint propagation. The vertices of this graph are literals, and each edge is labeled with the clause that forces the assignment. When a clause becomes unsatisfiable as a result of the current set of assignments (decision assignments or implied assignments), a conflict clause is introduced to record the cause of the conflict, so that the same futile search is never repeated. The conflict clause is learned from the structure of the implication graph. When the search backtracks, it backtracks to the most recent variable in the conflict clause just added, not to the variable that was assigned last. For our purposes, note that Equation 7 is unsatisfiable, and hence there will be much backtracking. Hence, many conflict clauses will be introduced before the SAT checker concludes that the formula is unsatisfiable. A conflict clause records a reason for the formula being unsatisfiable. The variables in a conflict clause are thus important for distinguishing between dead-end and bad states. The decision variable to which the search backtracks is responsible for the current conflict and hence is an important variable. We call the implication graph associated with each conflict a *conflict graph*.The source nodes of this graph are the variable decisions, the sink node of this graph is the conflicting assignment to one of the variables. At least one conflict clause is generated from a conflict graph. We propose the following two algorithms to identify important variables from conflict analysis and backtracking.

5.1 Refinement Based on Scoring Invisible Variables

We score invisible variables based on two factors, first, the number of times a variable gets backtracked to and, second, the number of times a variable appears in a conflict clause. Note that we have adjust the first score by an exponential factor based on the decision level a variable is at, as the variable at the root node can potentially get just two back tracks, while a variable at the decision level dl can get 2^{dl} backtracks globally. Every time the SAT procedure backtracks to an invisible variable at decision level dl, we add the following number to the *backtrack_score*.

$$2^{\frac{|\mathcal{I}|-dl}{c}}$$

We use c as a normalizing constant. For computing the second score, we just keep a global counter *conflict_score* for each variable and increment the counter for each variable appearing in any conflict clause. The method used for identifying conflict clauses from conflict graphs greatly affects SAT performance. As shown in [21], we use the most effective method called the *first unique implication point* (1UIP) for identifying conflict clauses. We then use weighted average of these two scores to derive the final score as follows.

$$w_1 \cdot \mathit{backtrack\_score} + w_2 \cdot \mathit{conflict\_score} \tag{8}$$

Note that the second factor is very similar to the decision heuristic VSIDS used in Chaff. The difference is that Chaff uses these per variable global scores to arrive at local decisions (of the next branching variable), while we use them to derive global information about important variables. Therefore, we do not periodically divide the variable scores as Chaff does.

We also have to be careful to guide Chaff not to decide on the intermediate variables introduced while converting various formulae to CNF form, which is the required input format for SAT checkers. This is done automatically in our method.

5.2 Refinement Based on Conflict Dependency Graph

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.

Dependencies Between Conflict Graphs We call the implication graph associated with a conflict a *conflict graph*. At least one conflict clause is generated from a conflict graph.

Definition 3. *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 B* ***directly depends*** *on conflict A.*

For example, consider the conflicts depicted in the conflict graphs of Figure 4. 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 [21] 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.

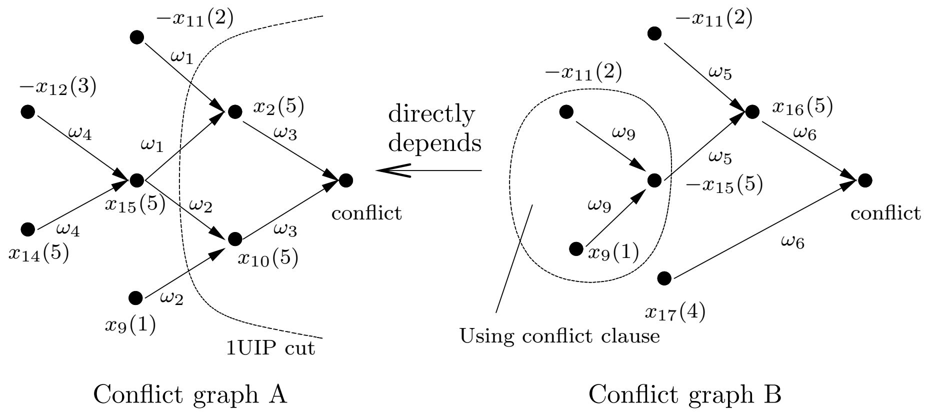


Fig. 4. Two dependent conflict graphs. Conflict B depends on conflict A, as the conflict clause ω_9 derived from the conflict graph A produces conflict B.

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

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

Figure 5 shows an unpruned 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 unpruned dependency graph. In Figure 5, 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. The subgraph of the unpruned conflict dependency graph on which the last conflict graph depends is called the *conflict dependency graph.* Formally,

Definition 4. *The* ***conflict dependency graph*** *is a subgraph of the unpruned dependency graph. It includes the last conflict graph and all the conflict graphs on which the last one depends.*

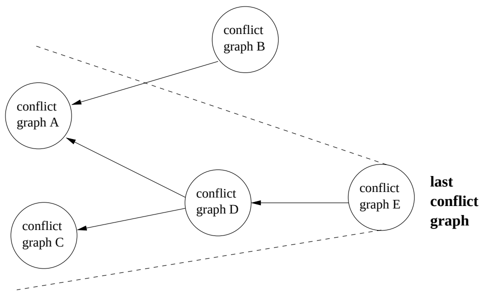


Fig. 5. The unpruned dependency graph and the dependency graph (within dotted lines)

In Figure 5, conflict graph E is the last conflict graph, hence the conflict dependency graph includes conflict graphs A, C, D, E. Thus, the conflict dependency graph can be constructed from the unpruned dependency graph by any directed graph traversal algorithm for reachability. Typically, many conflict graphs can be pruned away in this traversal, so that the dependency graph becomes much smaller than the unpruned 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.

Generating Conflict Dependency Graph Based on Zchaff We have implemented the conflict dependency analysis algorithm on top of zchaff [21], which has a powerful learning strategy called first UIP (1UIP). Experimental results from [21] 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 [21] for the details. We have built our algorithms on top of 1UIP, and we restrict the following discussions to the case that only one conflict clause is generated from a conflict graph. Note here that the algorithms can be easily adapted to other learning strategies.

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 pruned dependency graph.

Identifying Important Variables The dependency graph records the reasons for unsatisfiability. Therefore, only the variables appearing in the dependency graph are important. Instead of collecting all the variables appearing in any conflict graph, those in the dependency graph are sufficient to disable the spurious counterexample.

Suppose $\bar{s}_{f+1} = \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{f+1} \rangle$ is the shortest prefix of a spurious counterexample that can not be simulated on the concrete machine. Recall that $\hat{s}_f$ is the failure state. During the satisfiability checking of $\bar{s}_{f+1}$, we generate an unpruned conflict dependency graph. When Chaff terminates with unsatisfiability, we collect the clauses from the pruned conflict dependency graph. Some of the literals in these clauses correspond to invisible registers at time frame f. Only those portions of the circuit that correspond to the clauses contained in the pruned conflict dependency graph are necessary for the unsatisfiability. Therefore, the candidates for refinement are the invisible registers that appear at time frame f in the conflict dependency graph.

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 [19]. The algorithm in [19] has two phases. The first phase is the addition phase, where a set of invisible registers that it 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 identifed. 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. 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 our minimization algorithm considerably.

6 Experimental Results

We have implemented our abstraction refinement framework on top of NuSMV model checker [5]. We modified the SAT checker Chaff to compute heuristic scores, 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. All other experiments were performed on a dual 1.5GHz Athlon machine with 3GB of RAM running Linux.

The experiments were performed on two sets of benchmarks. The first set of benchmarks in Table 1 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 2 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 [8]. We include these results here to compare our methods with the results reported in [8] for property 2. We do not report the results for property 1 in [8] 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 1, we compare our methods against the BDD based model checker Cadence SMV. We enabled cone of influence reduction and dynamic variable reordering in Cadence SMV. 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 our heuristic variable scoring method. We introduce 5 latches at a time in this method. The columns labeled with "Dependency" report the results of our dependency analysis based refinement. This method employs pruning of candidate refinement sets. A "-" in a cell indicates that the model checker ran out of memory.

circuit	# regs	ctrex	CSMV	Heuristic Score			Dependency		
		length	time	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 1. Comparison between Candence SMV (CSMV), heuristic score based refinement and dependency analysis based refinement for larger circuits. The experiment marked with a * was performed on the SunFire machine with more memory because of a length 72 abstract counterexample encountered.

Table 2 compares our methods against those reported in [8] on IU series benchmarks for verifying property 2.

circuit	# regs	ctrex	[8]	Heuristic Score			Dependency		
		length	time	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 2. Comparison between [8], heuristic score based refinement and dependency analysis based refinement for smaller circuits.

We can see that our conflict dependency analysis based method outperforms a standard BDD based model checker, the method reported in [8] and the heuristic score based method. We also conclude that the computational overhead of our dependency analysis 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 dependency analysis. Therefore, it results in a coarser 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 2. 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).

7 Conclusions

We have presented an effective and practical automatic abstraction refinement framework based on our novel SAT based conflict analysis. We have described a simple variable scoring heuristic as well as an elaborate conflict dependency analysis for identifying important variables. Our schemes are able to handle large industrial scale designs. Our work highlights the importance of using SAT based methods for handling large circuits. We believe these techniques complement bounded model checking in that they enable us to handle true specifications effeciently.

An obvious extension of our framework is to handle all ACTL* formulae. We believe this can be done as in [9]. Further experimental evaluation will help us fine tune our procedures. We can also use circuit structure information to accelerate the SAT based simulation of counterexamples, for example, by identifying replicated clauses. We are investigating the use of the techniques described in this paper for software verification. We already have a tool for extracting a Boolean program from an ANSI C program by using predicate abstraction.

8 Acknowledgements

We would like to thank Ofer Strichman for providing us some of the larger benchmark circuits.

References

[1] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Proceedings of CAV'93*, pages 29–40, 1993.

[2] Armin Biere, Alexandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS, 1999.

[3] Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, Jim Kukula, Tom Shiple, Helmut Veith, and Dong Wang. Non-linear quantification scheduling in image computation. In *Proceedings of ICCAD'01*, pages 293–298, November 2001.

[4] Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, Jim Kukula, Helmut Veith, and Dong Wang. Using combinatorial optimization methods for quantification scheduling. In Tiziana Margaria and Tom Melham, editors, *Proceedings of CHARME'01*, volume 2144 of *LNCS*, pages 293–309, September 2001.

[5] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings of the International Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499. Springer, July 1999.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of CAV*, volume 1855 of *LNCS*, pages 154–169, July 2000.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[8] Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proceedings of CAV'02*, 2002. To appear.

[9] Edmund Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, 2002. To appear.

[10] Satyaki Das and David Dill. Successive approximation of abstract transition relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, 2001.

[11] Shankar G. Govindaraju and David L. Dill. Counterexample-guided choice of projections in approximate symbolic model checking. In *Proceedings of ICCAD'00*, San Jose, CA, November 2000.

[12] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *Proceedings of ICCAD'00*, November 2000.

[13] R. Kurshan. *Computer-Aided Verification of Co-ordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[14] J. Lind-Nielsen and H. Andersen. Stepwise CTL model checking of state/event systems. In N. Halbwachs and D. Peled, editors, *Proceedings of the International Conference on Computer Aided Verification (CAV'99)*, 1999.

[15] David E. Long. *Model checking, abstraction and compositional verification*. PhD thesis, Carnegie Mellon University, 1993. CMU-CS-93-178.

[16] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[17] Abelardo Pardo and Gary D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Proceedings of the Design Automation Conference (DAC'98)*, pages 457–462, June 1998.

[18] J. P. Marques Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. Technical Report CSE-TR-292-96, Computer Science and Engineering Division, Department of EECS, Univ. of Michigan, April 1996.

[19] Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the DAC*, pages 35–40, 2001.

[20] Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the Conference on Automated Deduction (CADE'97)*, pages 272–275, 1997.

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

SAT based Predicate Abstraction for Hardware Verification

Edmund Clarke Muralidhar Talupur Dong Wang

Carnegie Mellon University

Abstract. Predicate abstraction has emerged as one of the most promising abstraction techniques. It has been used to extract compact finite state models, which are amenable to the current model checking algorithms, from infinite state systems like software. However, there is little work on applying predicate abstraction for verifying large scale finite state (e.g., hardware) systems. One of the major obstacles is the inefficiency of the existing refinement algorithms. In this paper, we present two SAT based algorithms to refine the abstract model. During the abstraction refinement process, constraints are added to remove *spurious transitions* (the transitions in the abstract model that do not have any corresponding concrete transitions). Our first algorithm makes use of the conflict graphs generated by SAT solvers to make the added constraints as general as possible, thus making the abstract model more accurate. One nice feature of this algorithm is that it does not need to make any additional calls to SAT solvers once an abstract transition is determined to be spurious. Even after all the spurious transitions are eliminated, a counterexample might still be spurious. In this case a new predicate needs to be added to the abstract model. Our second algorithm generates a compact predicate that will eliminate the spurious counterexample. This algorithm too makes use of the conflict graphs to determine the important concrete variables that render the counterexample spurious. And then it creates a predicate over these concrete variables, which is added to the abstract model. Experiments over hardware designs with up to thousands of registers demonstrate the effectiveness of our methods.

1 Introduction

Abstraction refinement. Model checking [6] is a widely used automatic formal verification technique. Despite the recent advancements in model checking technology, its application is still limited by the state explosion problem. For model checking large real world systems abstraction is essential. For the abstraction to be conservative the *abstract model* should include all the behaviors of the given system. If the abstraction is conservative then the correctness of any universal temporal logic formula (e.g., $ACTL^*$) on the abstract model implies the correctness of the formula on the concrete model (This is refereed to as the preservation theorem). We consider only safety properties in this paper. However, a counterexample on the abstract model may not correspond to any real path, in which case it is called a *spurious* counterexample. To get rid of a spurious counterexample, the abstraction needs to be made more precise via refinement. *Counterexample guided abstraction refinement* (CEGAR) automates this procedure. It uses the spurious abstract counterexample to guide the refinement of the current abstraction, so that the counterexample is excluded from the refined abstract model. The above procedure repeats until the property is confirmed or refuted.

Predicate Abstraction. Predicate abstraction [1, 8, 7, 11–13], is a special case of conservative abstraction. In predicate abstraction a set of predicates $\{P_1, \ldots, P_m\}$, is identified from the concrete system and the property to be verified,. 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 $A = \{P_1, P_2, .., P_m\}$. 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 m 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 the 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, $\rho(s, \hat{s}) = \bigwedge_{1 \le j \le m} P_j(s) \Leftrightarrow B_j(\hat{s})$. We now define the *concretization function* γ, which maps a set of abstract states to the corresponding set of concrete states.

Formally, let $\hat{f}$ be a propositional formula over the abstract state variables, $\gamma(\hat{f}) = \hat{f}[B_j \leftarrow P_j]$. In predicate abstraction [13], the abstract initial states $\hat{S_0}$ and the abstract transition relation $\hat{R}$ are defined as

$$\hat{S_0} = \bigwedge\{\hat{Y_1} \mid S_0 \rightarrow \gamma(\hat{Y_1})\} \tag{1}$$

$$\hat{R} = \bigwedge\{\hat{Y} \rightarrow \hat{Y}' \mid (R \wedge \gamma(\hat{Y})) \rightarrow \gamma(\hat{Y}')\} \tag{2}$$

where $\hat{Y}$ ($\hat{Y_1}$) is an arbitrary conjunction (disjunction) of the literals over the current abstract state variables $\{B_1, \ldots, B_m\}$ and $\hat{Y}'$ is an arbitrary disjunction of literals over the next state variables $\{B'_1, \ldots, B'_m\}$. The abstract model built according to equations (1) and (2) 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 3.1). Since an approximate abstract model contains all the behaviors of the original concrete system, the preservation theorem still holds.

Motivation. For software model checking, 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 [6], from infinite state systems. Since hardware systems are finite state, model checking (or simpler forms of abstraction, e.g., localization reduction [9]) 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 (1) and (2) 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.

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. So, we use a heuristic similar to the one given in [1] 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) (The approximation to the set of abstract initial states can be similarly done for equation (1)). 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 3.1). 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 [7] 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 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 [7].

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.

Related work. SAT based localization reduction has been investigated in [2]. To identify important registers for refinement, SAT conflict dependency analysis is used. Their method is similar to our algorithm for reducing the support of the predicates. However, the two differ in the following ways. First, we have generalized SAT conflict dependency analysis to find the set of predicates which disables a spurious transition; while the algorithm in [2] only finds important registers. Second, in this paper, we present a projection-based SAT enumeration algorithm to determine a new predicate that can be used to refine the abstract model. Third, we approximate the most accurate abstract model by intentionally excluding certain implications; while in [2], approximation is achieved through pre-quantifying invisible variables during image computation. Finally, Our experimental results show significant improvement over the method in [2].

An algorithm to make the abstract model more accurate given a fixed set of predicates is presented in [7]. Given 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 the two algorithms together. Furthermore, the work in [7] does not consider the problem of introducing new predicates to refine the abstract model.

Existing refinement algorithms to compute new predicates use techniques such as syntactical transformations [11] or pre-image calculation [5, 13], etc. While our algorithm is based on SAT. They also neglect the problem of making the representation of the predicates compact. This could result in large predicates, which affects the efficiency of abstraction and refinement.

Outline of the paper. The rest of the paper is organized as follows. In Section 2 we describe conflict dependency analysis. We present our method for refining abstract transition relation in Section 3. In the same section a new method for identifying predicates is described. Section 4 has the experimental results. Section 5 concludes the paper.

2 SAT Conflict Dependency Analysis

In this section, we give a brief review of *SAT conflict dependency analysis* [2]. Modern SAT solvers rely on conflict driven learning to prune the search space. As presented in [14], a conflict clause corresponds to a vertex cut of a conflict graph (an implication graph with the conflict vertex as the sink), that separates the decision vertices from the conflict vertex. Let G be a conflict graph, κ be the conflict vertex in G, CUT be a vertex cut which corresponds to the conflict clause $cl(CUT)$. Let G_{CUT} be the subgraph of G where vertices in CUT are the sources and κ is the sink. For a subgraph G' of a conflict graph G, let $\Omega(G')$ be the set of clauses that label the edges in G'. Since G_{CUT} includes the conflict vertex κ, it is easy to see that $\neg cl(CUT) \wedge \Omega(G_{CUT}) \Rightarrow \text{false}$. Therefore

$$\Omega(G_{CUT}) \Rightarrow cl(CUT) \tag{3}$$

Given a CNF formula f, a SAT solver concludes that f is unsatisfiable if and only if the SAT solver derives a conflict graph without decision vertices. We associate the empty conflict clause, denoted by θ, with this last conflict graph. Note that since θ is an empty clause, it is logically equivalent to false.

A conflict clause $cl(CUT)$ *directly depends* on a clause b iff b is one of the clauses in $\Omega(G_{CUT})$. 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}. Given a CNF formula f, 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 $dep(cls)$. Based on equation (3), it is easy to see that $dep(cls) \Rightarrow cls$. If f is an unsatisfiable CNF formula, let $SUB(f) = dep(\theta)$. Since $dep(\theta) \Rightarrow \theta$, $SUB(f) \subseteq f$ is unsatisfiable, i.e.,

$$f \equiv \text{false} \Rightarrow SUB(f) \equiv \text{false} \tag{4}$$

During SAT search, our conflict dependency analysis algorithm keeps track of the set of clauses on which a conflict clause directly depends. After the SAT solver concludes that f is unsatisfiable, our algorithm identifies the unsatisfiable subset $SUB(f)$ based on these dependencies. Note that the dependencies and the unsatisfiable subset that our algorithm computes are determined by the conflict graphs and the conflict clauses generated by a SAT solver during SAT search. 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.

3 Refinement for Predicate Abstraction

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, $f(V) = utf(f^i(V^i))$. 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., $utr(r^i(V^i, V^{i+1})) = r(V, V^{'})$.

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

$$trans(i-1, i) = \gamma(ce_{i-1}) \wedge R^{i-1} \wedge \gamma(ce_i) \tag{5}$$

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:

$$prf(i) = S_0 \wedge \gamma(ce_0) \wedge R^0 \wedge \cdots \wedge \gamma(ce_{i-1}) \wedge R^{i-1} \wedge \gamma(ce_i). \tag{6}$$

Let BV be a set of boolean variables and let $BV_1 \subseteq BV$. If c is a conjunction of literals over BV, the projection of c to BV_1, denoted by $\texttt{proj}[BV_1](\texttt{c})$, is a conjunction of literals over BV_1 that agrees with c over the literals in BV_1. 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, $SA[BV_1](f) = \texttt{proj}[BV_1](SA[BV](\texttt{f}))$. For a SAT solver with conflict based learning, there is a well known algorithm to compute $SA[BV_1](f)$ without first computing $SA[BV](f)$ [10]. 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 3.1, 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 3.2 we deal with the case where the the set of predicates is not sufficient.

3.1 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 $trans(i-1, i) = R^{i-1} \wedge \gamma(ce_{i-1}) \wedge \gamma(ce_i)$ 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 $trans(i-1, i)$ 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 $R^{i-1} \wedge \gamma(ce_{i-1}) \wedge \gamma(ce_i) \Leftrightarrow \text{false}$. 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 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

$$F = \exists t_1, t_2 \ldots, t_{2m}. \ f \wedge \bigwedge_{j \in \{1, \ldots, 2m\}} (t_j \wedge (t_j \equiv f_j)). \tag{7}$$

It is easy to see that this formula is unsatisfiable iff the original formula is unsatisfiable. Once (7) 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 (7) is unsatisfiable, we know that $SUB(F) \subseteq F$ is unsatisfiable, where $SUB(F)$ is defined as in Section 2. It can be shown that $care = \{j \mid T_j \in SUB(F)\}$ represents the desired set of f_j's. Using the set $care$, we can add a more general constraint to $\hat{R}$.

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 [7], 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 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 [7], which requires $|care|$ additional calls to the SAT solver. Since the size of $care$ is already small, this is not very expensive.

3.2 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 $prf(k-1) \neq \emptyset \wedge prf(k) = \emptyset$. If there is no such k then the counterexample is real. Otherwise, the set of states $SA[V^{k-1}](prf(k-1))$ is called the set of *deadend states*, denoted by $deadend$ [5]. 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 $SA[V^{k-1}](trans(k-1,k))$ is called the set of *bad states*, denoted by bad [5]. The states in bad are those states in $\gamma(ce_{k-1})$ that have a transition to some state in $\gamma(ce_k)$. For a spurious abstract counterexample ce without spurious transitions, let k be the length of the spurious prefix of ce. Then $deadend \neq \emptyset$, $bad \neq \emptyset$ and $(deadend \cap bad) = \emptyset$. As is pointed out in [5], 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.

Our algorithm to compute the desired support is similar to the one used in finding the important registers in localization reduction in [2]. Since the CNF formula for $prf(k)$ is unsatisfiable, we can use conflict dependency analysis from Section 2 to identify $SUB(prf(k))$ that is unsatisfiable. Let all the concrete state variables at time $k-1$ whose CNF variables are in $SUB(prf(k))$ be $\mu(ce, k-1)$. That is $\mu(ce, k-1) = ssuppt[V^{k-1}](SUB(prf(k)))$. For the sake of brevity we will refer to $\mu(ce, k-1)$ as μ. Let $deadend_\mu = \texttt{proj}[\mu](deadend)$ be the projection of the deadend states on μ. Let $bad_\mu = \texttt{proj}[\mu](bad)$

be the projection of the deadend states on μ. It can be shown that

$$\mu \neq \emptyset \wedge deadend_{\mu} \cap bad_{\mu} \neq \emptyset. \tag{8}$$

Thus any concrete set of states S_1 that satisfies $(S_1 \supseteq deadend_{\mu}) \wedge (S_1 \cap bad_{\mu} = \emptyset)$ is a candidate separating predicate. To further reduce the size of μ and to make it minimal we use the refinement minimization algorithm in [2], which eliminates any unnecessary variables in μ while ensuring that equation (8) still holds. 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 first. 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 $SODE = \texttt{proj}[\mu](ODE \wedge bad)$ 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, an important register computed using algorithms in [2] 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 $\mu_1, \ldots, \mu_l$ based on the closeness of the variables (the criterion for closeness is based on circuit structure [3]). Because each set is small, we can compute each $deadend_{\mu_i}$ easily. The over-approximation is $ODE = \wedge deadend_{\mu_i}$.

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. It can be shown that the concrete transition relation implies the concretization of this constraint. Therefore, the spurious counterexample is invalidated in the refined abstract model.

4 Experimental Results

We have implemented our predicate abstraction refinement framework on top of NuSMV model checker [4]. We modified the SAT checker zChaff [14] to support conflict dependency analysis. We also developed a Verilog parser to extract useful predicates from the Verilog design directly. We do not go into the details of the parser due to lack of space. 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 Microsystems; 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. For all the properties shown in the first column of Table 1, we have performed cone-of-influence reduction before the verification. The resulting number of registers and gates are shown in the second and third columns. We compare three abstraction refinement systems, including the BDD based aSMV [5], the SAT based localization reduction [2] (SLOCAL), and the SAT based predicate abstraction (SPRED) described in this paper. The detailed results obtained using aSMV are not listed in Table 1 because aSMV can not solve any of the properties within the 24hr time limit. This is not surprising because aSMV uses BDD based image computation and it can handle only circuits with hundreds of state variables, provided that good initial variable orderings are given. Since the time to generate good BDD variable orderings can be substantial, we did not pre-generate them for any of

circuit	# regs	# gates	ctrex length	Localization			Predicate Abstraction		
				time	iters	# regs	time	iters	# predicates
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 1. Comparison between localization reduction [2] and predicate abstraction.

the properties. For the first four properties from IU, SLOCAL takes about twice the time taken by SPRED. Furthermore, the numbers of registers in the final abstract models from SLOCAL are much larger than the corresponding numbers 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.

5 Conclusion

We have presented two SAT based counterexample guided refinement algorithms to enable efficient predicate abstraction of hardware designs with up to thousands of registers. To reduce the abstraction time, an approximate abstract model is built initially, which could result in *spurious transitions*. Once a spurious transition is identified from a given abstract counterexample using SAT, our first SAT based refinement algorithm eliminates this transition (and possibly many other related spurious transitions) without any additional calls to a SAT solver. An abstract model may also fail to determine the result of verification when the generated abstract counterexample has a spurious prefix. To eliminate a spurious prefix, our second SAT based refinement algorithm can compute a new predicate with the minimal number of supporting concrete state variables. Usually, the predicates our algorithm computes can be represented compactly as a CNF formula or a conjunction of DNF formulas. Experimental results show significant improvement of our predicate abstraction algorithms over popular abstraction algorithms for hardware verification.

References

1. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI 2001*.
2. 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.
3. 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.
4. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In *CAV'99*, pages 495–499, 1999.
5. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *CAV'00*, 200.
6. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
7. S. Das and D. Dill. Successive approximation of abstract transition relations. In *LICS'01*, 2001.
8. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *CAV'99*, pages 160–171, 1999.
9. R. P. Kurshan. *Computer-Aided Verification*. Princeton Univ. Press, Princeton, New Jersey, 1994.
10. K. McMillan. Applying sat methods in unbounded symbolic model checking. In *CAV'02*, pages 250–264, 2002.
11. K. Namjoshi and R. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV'00*, 2000.
12. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV'97*, pages 72–83, 1997.
13. H. Saidi and N. Shankar. Abstract and model check while you prove. In *CAV'99*, pages 443–454, 1999.
14. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *ICCAD'01*, 2001.

High Level Verification of Control Intensive Systems Using Predicate Abstraction [*]

Edmund Clarke
Carnegie Mellon Univ.
Pittsburgh, PA 15217
emc@cs.cmu.edu

Orna Grumberg
TECHNION
Technion City, Haifa 32000, Israel
orna@cs.technion.ac.il

Muralidhar Talupur, Dong Wang
Carnegie Mellon Univ.
Pittsburgh, PA 15217
{tmurali,dongw}@cs.cmu.edu

Abstract

Predicate abstraction has been widely used for model checking hardware/software systems. However, for control intensive systems, existing predicate abstraction techniques can potentially result in a blowup of the size of the abstract model. We deal with this problem by retaining important control variables in the abstract model. By this method we avoid having to introduce an unreasonable number of predicates to simulate the behavior of the control variables. We also show how to improve predicate abstraction by extracting useful information from a high level representation of hardware/software systems. This technique works by first extracting relevant branch conditions. These branch conditions are used to invalidate spurious abstract counterexamples through a new counterexample-based lazy refinement algorithm. Experimental results are included to demonstrate the effectiveness of our methods.

1 Introduction

Background. Abstraction based model checking has been widely accepted as a valuable method for the verification of large hardware/software systems. *Predicate abstraction* [1, 2, 3, 10, 11, 13, 16, 18, 19], in particular, is one of the most successful abstraction techniques. In predicate abstraction, the concrete system is approximated by only keeping track of certain predicates over the concrete state variables. Each predicate corresponds to an abstract boolean variable. Any concrete transition corresponds to a change of values for the set of predicates and is subsequently translated into an abstract transition. Using predicate abstraction, it is possible to not only reduce the complexity of the system under verification, but also, for software systems, to extract finite models that are amenable to model checking algorithms.

Predicate abstraction is a special case of existential abstraction [6, 9, 15], which is a conservative approach for model checking universal temporal logic [9] properties (we only consider safety properties in this paper). 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 [7]. To get rid of a spurious counterexample, the abstraction needs to be made more precise via refinement. *Counterexample guided abstraction refinement* [7, 13, 20] (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.

Motivation. 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/software systems which contain extensive control structure (*control intensive systems*). The control logic usually consists of concurrent state machines.

[*]This research is sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, 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 SRC, GSRC, NSF, or the United States Government.

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.

Another issue that we address in this paper is the following: Current predicate abstraction methods do not make use of information available in the high level descriptions of the system under verification. Most hardware/software codesign tools use high level design languages, such as ESTEREL, graphical FSMs, RTL Verilog/VHDL, C/C++ 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 it 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.

Performance. Experiments we performed demonstrate the efficacy of our methods. In one series of experiments, the current predicate abstraction methods could not verify the given properties even after 24 hrs, whereas, our method could verify the same properties in less than 15 mins.

Outline of the paper. In the next section we introduce predicate abstraction and other relevant theory. In Section 3, we give a clustering based heuristic to identify control variables and present a modified localization reduction algorithm to bound the size of the abstract model. Algorithms to compute accurate abstract models are also discussed in the same section. Section 4 gives the predicate extraction and refinement algorithm. Some related work is discussed in Section 5. In Section 6, we describe our experiments. Section 7 concludes the paper.

2 Preliminary

In this section, we review the theory of *existential abstraction*. We then present *predicate abstraction* and the *localization reduction* as special cases of this concept.

2.1 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* (refer to [9] for details).

Definition 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 $M \equiv_{bis} \hat{M}$.

Theorem 2.1 *(Preservation of ACTL* [9])*
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* $A \cap \hat{A}$

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

In the above theorem, if ρ is a bisimulation relation, then for any CTL* formula Φ with atomic propositions in $A \cap \hat{A}$, $\hat{M} \models \Phi \Leftrightarrow M \models \Phi$.

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 $s \in S$ that satisfy $\rho(s, \hat{s})$ 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:

$$\hat{S_0} = \exists s.\ S_0(s) \wedge \rho(s, \hat{s}) \qquad (1)$$

$$\hat{R}(\hat{s}, \hat{s}') = \exists s\ s'.\ \rho(s, \hat{s}) \wedge \rho(s', \hat{s}') \wedge R(s, s') \qquad (2)$$

$$\text{for each } \hat{s} \in \hat{S},\ \hat{L}(\hat{s}) = \bigcap_{\rho(s,\hat{s})} (L(s) \cap \hat{A}) \qquad (3)$$

This kind of abstraction is called *existential abstraction* [6, 15].

2.2 Predicate Abstraction

Predicate abstraction 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,

$$\rho(s, \hat{s}) = \bigwedge_{1 \leq j \leq k} P_j(s) \Leftrightarrow B_j(\hat{s}) \qquad (4)$$

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. Using this ρ and the construction given in the previous subsection, we can build an abstract model which simulates the concrete model. We now define the *concretization function* γ, which maps a set of abstract states to the corresponding set of concrete states. Formally, let $\hat{f}$ be a propositional formula over abstract state variables,

$$\gamma(\hat{f}) = \hat{f}[B_j \leftarrow P_j]. \qquad (5)$$

In predicate abstraction [19], the abstract initial states $\hat{S_0}$ and the abstract transition relation $\hat{R}$ are defined as

$$\hat{S_0} = \bigwedge\{\hat{Y_1} \mid S_0 \rightarrow \gamma(\hat{Y_1})\} \qquad (6)$$

$$\hat{R} = \bigwedge\{\hat{Y} \rightarrow \hat{Y}' \mid (R \wedge \gamma(\hat{Y})) \rightarrow \gamma(\hat{Y}')\} \qquad (7)$$

where $\hat{Y}$ ($\hat{Y_1}$) is an arbitrary conjunction (disjunction) 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 $\{B'_1, B'_2, \ldots, B'_k\}$. It can be shown that (6) is equivalent to (1) and (7) is equivalent to (2).

Equations (6) and (7) 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 [17], is used. For hardware verification, a SAT solver, such as zChaff, can be more efficient. In practice, heuristics are used to reduce the number of calls to the theorem prover [1, 19]. In this paper, 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 Localization Reduction

Localization reduction [14] 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 dropped (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. The value of a variable $v \in V$ in state $s \in S$ is denoted by $s(v)$. Given a set of variables $U = \{u_1, u_2, \ldots, u_k\}, U \subseteq V$, let s^U denote the portion of s that corresponds to the variables in U, i.e., $s^U = (s(u_1)s(u_2)\ldots s(u_k))$. Let U be the set of visible variables. 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 $\rho(s, \hat{s}) = (s^U \equiv \hat{s})$.

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. Thus the abstract initial states $\hat{S_0}$ and the abstract transition relation $\hat{R}$ are defined as

$$\hat{S_0} = \wedge_{v \in U} I_v \qquad (8)$$

$$\hat{R} = \wedge_{v \in U} R_v \qquad (9)$$

It is usually the case that $\hat{R}$ depends not only on current and next state variables on U, but also some invisible variables (precisely those invisible variables that occur in some R_v or I_v). In the abstract model, these invisible variables are treated as primary inputs. In general, the abstract model for localization reduction can be computed very easily, but the size of the abstract transition relation may be large since it is directly copied from the concrete model.

3 Clustering Based Predicate Abstraction

In this section, we show how to use clustering based heuristics to identify control variables. We present an algorithm to build an abstract model by combining localization reduction with predicate abstraction. This procedure ensures that the size of the abstract model is bound by the size of the concrete model. We also show how to use correlations between predicates and control variables to make the abstract model more accurate.

3.1 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 $v(c)$. If all the variables in $v(c)$ are finite state, each variable can be represented by 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.

3.2 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 (6) and (7) 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 most 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 circumvent the problem of building abstract transition relations for all these control variables.

The modification to the concrete transition relation is as follows: for a supporting variable $v \in v(c)$, let R_v be the concrete transition relation for v. Let $R'_v = R_v[P_j \leftarrow B_j$, for all j such that P_j 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. If R'_v is finite state, that is, if there are no unbounded variables or unbounded control (e.g., recursion) in it, then we use v as an abstract state variable. In such a case 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. Doing this has two advantages: Firstly, even if R_v had unbounded variables, R'_v could be finite state because of the substitutions. Secondly, 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. This can increase the size of the abstract model. Therefore, we retain unabstracted only those variables whose next state logic has a small number of inputs.

3.3 Correlations between Control Variables and Predicates

Our abstract model includes real predicates and control variables. In this subsection, a method to correlate predicates and control variables will be discussed. Recall from Section 3.1 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 $x = \mathrm{false}$ and $y = \mathrm{false}$ in that state.
- If we know $x = \mathrm{false}$ 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. Note that, only formula predicates whose supporting variables are all finite state are considered in this method. 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.

3.4 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 paragraph, when $B_3 = \mathrm{false}$, 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

$$g(c) = \{\hat{s} \mid \exists s \in S. \bigwedge_{j \in I(c)} (P_j(s) = B_j(\hat{s}))\} \tag{10}$$

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 $g(c)$. We have two algorithms depending on whether or not there are any unbounded variables in $v(c)$. The first algorithm is based on BDDs. It only works if all variables in $v(c)$ have finite domains. We can 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. The second algorithm is based on the abstraction function [19]. Let $\hat{Y}(c)$ be a disjunction of literals over variables B_j, where $j \in I(c)$. It can be shown that $g(c)$ is the same as

$$\bigwedge\{\hat{Y}(c) \mid \mathrm{true} \Rightarrow \gamma(\hat{\mathrm{Y}}(\mathrm{c}))\} \tag{11}$$

Essentially, this equation says that a formula over the abstract variables, $\hat{Y}(c)$, includes the set of consistent abstract states if the corresponding concrete formula, $\gamma(\hat{Y}(c))$, is true. The second algorithm works for variables with both finite and infinite domains. For the finite case, a SAT solver can be used; while for the other case, a general theorem prover has to be used. Since the second algorithms may require solving $\mathrm{true} \Rightarrow \gamma(\hat{\mathrm{Y}}(\mathrm{c}))$ for all possible disjunctions over variables in cluster c, it is usually slower than the first algorithm when variables have finite domains.

4 Exploiting High Level Representation

In this section, we discuss how to improve predicate abstraction by using information from the high level representation of the design under verification. We first describe our method for extracting branch conditions from RTL Verilog and then we present our lazy-refinement algorithm to refine the abstract model.

4.1 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 [12]. In practice Verilog is interpreted by a set of standard commercial tools, such as *Synopsys Design Compiler*. 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 1. We use the *continuous assignment statement* in Verilog to rename the branch conditions using unique signals, such that the modified design is equivalent to the original one. After this,

```
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 1. Replace branch conditions using unique signals

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 4.2 to identify a subset of the branch conditions which are necessary to invalidate the given spurious abstract counterexample.

4.2 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, our algorithm 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'.

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 2.2. 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 2 illustrates how our algorithm works. Suppose there are 2 predicates, 3 candidate predicates 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

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

Figure 2. A refinement example

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

$$(R \wedge \gamma(ce_0) \wedge cp_3 \wedge \overline{cp_4} \wedge \overline{cp_5} \wedge \gamma(ce_1)) \rightarrow cp_3' \quad (12)$$

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

$$(R \wedge \gamma(ce_1) \wedge cp_3 \wedge cp_5) \rightarrow \gamma(\overline{ce_2}) \quad (13)$$

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 (12) and (13) are necessary in showing that they are tautologies. Only those in the proof of the tautologies are necessary. Proofs can be obtained from *proof generating theorem provers* (e.g., Simplify) and *proof generating SAT solvers* [4]. Suppose, we can determine that cp_3, cp_5 in equation (12) and cp_5 in equation (13) 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 (13)). 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 (12)). 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 3. The second one, which computes ece_{i+1} making use of ece_i, is shown in Figure 4. The last one, shown in Figure 5, 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 3. Algorithm to compute ece_0

```
    //$i$: 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 4. Algorithm to compute ece_{i+1}

```
   //$t$: 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. Algorithm to compute invalidating predicates

The algorithm to compute ece_0 is similar to the algorithm for computing the set of abstract initial states in Section 2.2, 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 4 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 to compute a new predicate [7] using SAT. Details can be found in [8].

If the counterexample is invalidated at line (1) in Figure 4, the algorithm in Figure 5 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 4. 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 Related Work

Some researchers have considered combining unabstracted control variables with predicate abstraction [16], but their methods are not automatic. As far as we know, no one else has considered the correlation between unabstracted control variables and predicates. Using the correlations between all predicates to constrain the abstract model has been investigated in [1]. 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 for the verification of finite state systems.

Exploiting high level language features for abstraction has been investigated in [7]. They extract conditions of **case** statements in the SMV language in order to build the initial abstraction. The extraction method in [7] requires modifying the *source code* of an existing translator from SMV language to transition relations, therefore it can not be applied to commercial tools. The extracted conditions are used only for the initial abstraction; while we use a new refinement algorithm to check whether branch conditions can invalidate the spurious abstract counterexample. The branch conditions become predicates only when they invalidate a spurious counterexample.

Our counterexample-based lazy refinement algorithm tries to identify the branch conditions that can invalidate the spurious abstract counterexample, before using the traditional refinement methods to compute a new predicate. Therefore, our algorithm is an extension of the existing refinement algorithms. Our experiments show that this new refinement algorithm can identify the set of predicates to verify the given property much more quickly than the traditional methods alone.

Lazy abstraction for the verification of C programs has been investigated in [13]. The goals of their algorithm and ours are different. In [13], 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 refinement algorithm identifies a subset of the branch conditions that can invalidate a spurious counterexample without constructing the full refined abstract model.

6 Experimental Results

We have implemented our predicate abstraction refinement framework in NuSMV [5]. We also modified the zChaff SAT solver [21] to generate proofs of unsatisfiability. We have two sets of 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. The size of the benchmarks is shown in Table 1. The first column is the name of the property. The first three properties are from the IU design; the remaining six are from the PFIR design. For all the properties shown in the first column of Table 2, we have performed cone-of-influence reduction before the verification. The resulting number of registers and gates are shown in the second and third columns. Most properties are true, except PFIRscr1 and PFIRprop5. The lengths of the counterexamples are shown in the fourth column.

circuit	# regs	# gates	ctrex
IUscr1	4855	149143	true
IUscr3	4855	149143	true
IUscr6	4855	149143	true
PFIRscr1	243	2295	16
PFIRprop5	250	2342	17
PFIRprop8	244	2304	true
PFIRprop9	244	2304	true
PFIRprop10	244	2304	true
PFIRprop12	247	2317	true

Table 1. The benchmarks used in the experiments

All these properties are difficult for the state-of-art BDD-based model checker, Cadence SMV. Except for the two false properties, Cadence SMV can not verify any in 24 hours. The verification time for PFIRscr1 is 834 seconds, and for PFIRprop5 is 8418 seconds. In Table 2, we compare predicate abstraction with and without the techniques presented in this paper. In Table 2, the second to fourth columns are the results obtained without our techniques; while the last three columns are the results obtained with the techniques enabled. We compare the time (in seconds), the number of refinement iterations and the number of predicates in the final abstraction. In all cases, our new method outperforms the old one in the amount of time used; sometimes over an order of magnitude improvement is achieved. In most cases, we use fewer refinement iterations and smaller predicate sets to verify the given properties. 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 3 properties).

circuit	Old			New		
	time	iters	pred	time	iters	pred
IUscr1	2000	11	7	1265	7	18
IUscr3	2003	10	6	1974	16	7
IUscr6	9976	27	12	3498	20	11
PFIRscr1	746	109	44	386	67	34
PFIRprop5	1616	110	43	756	101	44
PFIRprop8	>24h	>276	>80	159	40	25
PFIRprop9	>24h	>189	>47	202	43	27
PFIRprop10	6808	170	52	178	50	25
PFIRprop12	>24h	>223	>52	591	80	38

Table 2. Comparison without and with our techniques

7 Conclusion

We have presented two techniques to improve predicate abstraction for the verification of hardware/software systems. We give an algorithm based on localization reduction to avoid the potential blowup of the abstract models when verifying control intensive systems. This technique builds a "hybrid" abstract model, which includes predicates as well as unabstracted control variables. It is usually the case that the predicates/control variables are not independent. We give algorithms to compute correlations between them, which help to make the abstract model more accurate. We also present algorithms to exploit information in high level design languages. We give a simple method to extract branch conditions from high level design representations. Using a new counterexample-based lazy refinement algorithm, the necessary branch conditions can be added as new predicates to invalidate spurious abstract counterexamples. Experimental results demonstrate the usefulness of our methods.

References

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

[2] 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.

[3] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Computer-Aided Verification, CAV'98*, pages 319–331, 1998.

[4] 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.

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

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

[7] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *CAV'00*, 200.

[8] Edmund Clarke, Muralidhar Talupur, and Dong Wang. SAT based Predicate Abstraction for Hardware Verification. Technical Report CMU-ECE-CSSI 02-45, Carnegie Mellon University, ECE Department, 2002.

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

[10] Michael Colon and Tomas E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV'98*, pages 293–304, 1998.

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

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

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

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

[15] 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.

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

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

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

[19] H. Saidi and N. Shankar. Abstract and model check while you prove. In *CAV'99*, pages 443–454, 1999.

[20] 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.

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

Verification of Hybrid Systems Based on Counterexample-Guided Abstraction Refinement

Edmund Clarke[1], Ansgar Fehnker[2], Zhi Han[2], Bruce Krogh[2], Olaf Stursberg[2,3], and Michael Theobald[1]

[1] Computer Science, Carnegie Mellon University, Pittsburgh, PA
[2] Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA
[3] Process Control Lab, University of Dortmund, Germany

Abstract. Hybrid dynamic systems include both continuous and discrete state variables. Properties of hybrid systems, which have an infinite state space, can often be verified using ordinary model checking together with a finite-state abstraction. Model checking can be inconclusive, however, in which case the abstraction must be refined. This paper presents a new procedure to perform this refinement operation for abstractions of infinite-state systems, in particular of hybrid systems. Following an approach originally developed for finite-state systems [1, 2], the refinement procedure constructs a new abstraction that eliminates a counterexample generated by the model checker. For hybrid systems, analysis of the counterexample requires the computation of sets of reachable states in the continuous state space. We show how such reachability computations with varying degrees of complexity can be used to refine hybrid system abstractions efficiently. A detailed example illustrates our counterexample-guided refinement procedure. Experimental results for a prototype implementation of the procedure indicate its advantages over existing methods.

1 Introduction

Hybrid systems are formal models that include both continuous and discrete state variables. With the increasing use of hybrid systems to design embedded controllers for complex systems such as manufacturing processes, automobiles, and transportation networks, there is an urgent need for more powerful analysis tools, especially for safety critical applications. Tools developed so far for automated analysis of hybrid systems are restricted to low-dimensional continuous dynamics [3]. The reason for this limitation is the difficulty of representing and computing sets of reachable states for continuous dynamic systems. Recent publications have proposed two general approaches to deal with the complexity of hybrid system analysis, namely, modular analysis (e.g., [4, 5]) and abstraction (e.g., [6–8]). This paper focuses on the latter approach.

Abstraction maps a given model into a less complex model that retains the behaviors of interest [6]. In the context of hybrid system verification, abstraction transforms the inherently infinite state system into a finite-state model [7, 8]. Existing tools often do not consider the property itself when building an abstract model. Rather, an abstract representation is constructed for the entire hybrid system using a degree of detail which seems to be appropriate. If the abstraction is not appropriate to analyze the property, the whole abstraction process is started again, or the abstract model is globally refined [9].

As an alternative, we suggest a procedure that (a) starts from a coarse abstract model and a safety property, (b) identifies parts of the hybrid system which potentially violate the property, and (c) iteratively refines the abstract model until verification reveals whether or not the property in question is satisfied. A framework that follows this general scheme of abstraction, refinement, and analysis, is *counterexample-guided abstraction refinement (CEGAR)* [1, 10, 2]: For a given system the initial abstraction leads to a conservative model that is guaranteed to include all behaviors of the original system. Model checking is then applied to the abstract model. If the property is violated, the model checker produces a *counterexample* as an *execution path* for the abstract model for which the property is not true. If the counterexample corresponds to a behavior of the original system, then the property does not hold for the original system. Otherwise, the information provided by the counterexample is then used to *refine* the abstract model, i.e., some detail is added to the abstract model in order to obtain a more accurate, yet conservative, representation of the original model. In particular, the refined model is constructed so that it is guaranteed to exclude the *spurious* counterexample. The procedure of alternating between model checking and refinement is continued until the property is confirmed or refuted.

This procedure has recently been applied successfully to finite discrete systems in a variety of domains, particularly for the verification of digital circuits [1, 10]. Earlier work that is based on the use of counterexamples includes the localization reduction in the context of concurrent systems [2], and recent work has applied the technique to the verification of C-programs [11, 12].

This paper makes two important contributions. First, we extend counterexample-guided model refinement to *infinite-state* systems. Second, we show how our new approach can be applied to hybrid systems, which include both continuous and discrete state variables and thus have an infinite-state space. We provide effective means of coping with the difficulties of computing reachable sets for infinite state systems. In particular, we employ reachable set computations with varying degrees of complexity to refine hybrid system abstractions efficiently. This flexibility cannot easily be achieved with other verification tools for hybrid systems. We note that using counterexamples to guide generation of discrete abstractions is being pursued independently by Alur et al. at University of Pennsylvania.

The paper is structured as follows. Section 2 presents preliminaries on abstraction and counterexample-guided refinement. In Section 3 we describe a new verification approach that refines abstract models of infinite state systems based on counterexamples. We introduce hybrid systems in Section 4, and apply our new verification approach to hybrid systems in Section 5. Section 6 presents conclusions.

2 Preliminaries

We introduce the notions of abstraction and counterexample-guided refinement in a general setting for infinite state systems. The type of model we are working with throughout the section is a transition system defined as follows:

Definition 1 *Transition System.* A *transition system* is a 3-tuple $TS = (S, S_0, E)$ with a (possibly infinite) state set S, an initial set $S_0 \subset S$, and a set of transitions $E \subset S \times S$. $\diamond$

Given two transition systems A and C, A is said to be an *abstract model* of C if the following relation can be established.

Definition 2 *Abstraction.* A transition system $A = (\hat{S}, \hat{S}_0, \hat{E})$ with a finite set of states $\hat{S}$ is an *abstract model* of a transition system $C = (S, S_0, E)$, denoted $A \succeq C$, if there exists an *abstraction function* $\alpha : S \to \hat{S}$ such that:

– the initial set is $\hat{S}_0 = \{\hat{s}_0 | \, \exists s_0 \in S_0 : \hat{s}_0 = \alpha(s_0)\}$
– and $\hat{E} \supseteq \{(\hat{s}_1, \hat{s}_2) | \, \exists s_1, s_2 \in S \ : (s_1, s_2) \in E, \hat{s}_1 = \alpha(s_1), \hat{s}_2 = \alpha(s_2)\}$. $\diamond$

Sometimes the term *simulation* is used in the literature to describe the abstraction relation. In contrast to the definitions of abstraction in [1, 10], Defn. 2 allows that A includes *spurious transitions*, i.e., the set $\hat{E}$ may contain elements that do not correspond to transitions in C. As a consequence the abstraction function in Defn. 2 does not uniquely define A. Spurious transitions arise in the construction of abstractions of hybrid systems because in most cases sets of reachable states for continuous systems can not be represented and computed exactly.

Abstract models will be used to analyze properties of a given transition system. Throughout the paper, we will call the given system C the *concrete system*.

In order to construct a more detailed model from a given abstract model, we define the following concept of *model refinement*.

Definition 3 *Refinement of Abstract Models.* Given a concrete system $C = (S, S_0, E)$ and an abstract model $A = (\hat{S}, \hat{S}_0, \hat{E})$ such that $C \preceq A$, with abstraction function $\alpha : S \to \hat{S}$, a model $A' = (\hat{S}', \hat{S}'_0, \hat{E}')$ is called a *refined abstract model of* C *with respect to* A if two abstraction functions $\alpha' : S \to \hat{S}'$ and $\alpha'' : \hat{S}' \to \hat{S}$ exist, i.e., $C \preceq A' \preceq A$. $\diamond$

The property is verified for the concrete model C using an abstract model A. In this paper we will consider the verification of safety properties, defined as follows.

Definition 4 *Safety.* Given a transition system $TS = (S, S_0, E)$, let the set $B \subset S$ specify a set of *bad states* such that $S_0 \cap B = \emptyset$. We say that TS *is safe with respect to* B, denoted by $TS \models \mathbf{AG}\neg B$ iff there is no path in the transition system from an initial state in S_0 to a bad state in B. Otherwise we say TS *is unsafe*, denoted by $TS \not\models \mathbf{AG}\neg B$. $\diamond$

Definition 5 *Counterexamples.* A path $\sigma = (s_0, s_1, \ldots, s_m)$ of $TS = (S, S_0, E)$ with $s_m \in B$ is called a *counterexample* of TS with respect to the safety property $TS \models \mathbf{AG}\neg B$. Given a concrete transition system C, an abstract transition system A, and a counterexample σ in C, we say that $\hat{\sigma} = (\hat{s}_0, \hat{s}_1, \hat{s}_2, \ldots, \hat{s}_m)$ is the *corresponding abstract counterexample* of the abstract system A, if $\hat{s}_i = \alpha(s_i)$ holds for all $i \in \{0, \ldots, m\}$. Given a counterexample $\hat{\sigma}$ of A, σ is called a *corresponding concrete counterexample* if $\hat{s}_i = \alpha(s_i)$ and $(s_i, s_{i+1}) \in E$. If a counterexample $\hat{\sigma}$ of A has no corresponding concrete counterexample for C, $\hat{\sigma}$ is called a *spurious counterexample*.$\diamond$

Lemma 1. *Given a concrete model* $C = (S, S_0, E)$*, and an abstract model* $A = (\hat{S}, \hat{S}_0, \hat{E})$ *of* C *with an abstraction function* α*, let* $B \subseteq S$*, and* $\hat{B} = \{\hat{b} \mid \exists \, b \in B : \hat{b} = \alpha(b)\}$*. If* $A \models \mathbf{AG}\neg\hat{B}$*, then* $C \models \mathbf{AG}\neg B$*.* $\square$

If $A \models \mathbf{AG}\neg\hat{B}$ can be verified, it can immediately be concluded from Lemma 1 (i.e., without applying verification to the concrete system C) that $C \models \mathbf{AG}\neg B$. On the other hand, the converse of Lemma 1 with respect to the **AG**-property is not possible. If the verification of A reveals $A \not\models \mathbf{AG}\neg\hat{B}$, then we cannot conclude that C is not safe with respect to B, since the counterexample for A may be spurious. We call a method that checks whether or not a counterexample is spurious a *validation method*. If the validation method discovers that the counterexample is spurious, then the counterexample is used to refine A. We now introduce a scheme for *counterexample-guided refinement of abstractions* to verify safety properties for a given concrete model. The basic principle is to repeat the following sequence of steps until the property is verified or refuted [1]. The starting point is a concrete model C and an abstract model A (we propose in Sec. 5.1 one specific way to obtain an initial abstract model for hybrid systems). For a set $B \subseteq S$ of bad states for C, we assume for simplicity that $\alpha(s) \in \hat{B}$ implies $s \in B$. The first step is then to analyze $A \models \mathbf{AG}\neg\hat{B}$ by model checking. If this property holds it can immediately be concluded from Lemma 1 that C is safe, too. Otherwise a counterexample is obtained, and it must be validated whether it has a corresponding counterexample in C. If there is a corresponding counterexample in C, then the safety property does not hold for C. In the other case, i.e. the counterexample is spurious, the counterexample is used to refine the model A. That is, a new and more detailed model A' with $C \preceq A' \preceq A$ is determined, which excludes the spurious counterexample.

The procedure of model checking, validation of the counterexample, and refinement of the abstract model is repeated until the safety property is proved or refuted for C. The pseudo-code in Fig. 1 summarizes this procedure:

```
ALGORITHM: Counterexample-Guided Abstraction Refinement: CEGAR
INPUT: Concrete model $C$ and a set of bad states $B$
OUTPUT: $B$ is (or is not) reachable

Generate initial abstract model $A$ (bad states are called $\hat{B}$)
Generate counterexample $\hat{\sigma}$ by model checking $A$ wrt. $\hat{B}$
WHILE $\hat{\sigma}$ exists DO
        Validation of $\hat{\sigma}$
        IF $\hat{\sigma}$ validated THEN terminate with "B reachable"
        ELSE
            Generate refined model $A'$ using counterexample $\hat{\sigma}$
            $A := A'$
            Generate next $\hat{\sigma}$ by model checking $A$ wrt. $\hat{B}$
        ENDIF
ENDDO
Terminate with "B not reachable"
```

Fig. 1. CEGAR: Scheme for verifying/falsifying $C \models \mathbf{AG}\neg B$ based on counterexample-guided abstraction refinement

The crucial steps in the CEGAR procedure are *validation*, *refinement*, and *model checking*. With respect to model checking, standard algorithms for AG-properties can be used [13].

The important step in validating a counterexample is the computation of successors of states. We define an operator $succ$ that determines the successor states from a given set $\tilde{S} \subseteq S$ by $succ(\tilde{S}) = \{s \in S | \exists \tilde{s} \in \tilde{S} : (\tilde{s}, s) \in E\}$. This set may not be exactly computable for a given concrete model C, i.e. only over-approximations $\overline{succ}(\tilde{S}) \supset succ(\tilde{S})$ may be available. We first assume that $succ(\tilde{S})$ is computable.

A counterexample $\hat{\sigma} = (\hat{s}_0, \ldots, \hat{s}_m)$ of A is then validated as follows: Let $S_k = \alpha^{-1}(\hat{s}_k)$, $k \in \{0, \ldots, m\}$ denote the set of concrete states corresponding to an element of $\hat{\sigma}$. The reachable parts of these sets are recursively defined by $S_0^{reach} := S_0$,

$S_k^{reach} := succ(S_{k-1}^{reach}) \cap S_k$, $k \in \{1, \ldots, m\}$. The counterexample is spurious iff $S_k^{reach} = \emptyset$ applies for at least one k, and we say *the counterexample is refuted*. Otherwise, the counterexample is *validated*, and B is reachable.

If the counterexample is refuted with $S_k^{reach} = \emptyset$, the model A is refined to a new finite abstract model $A' = (\hat{S}', \hat{S}'_0, \hat{E}')$ (cf. Defn. 3). The refined model should take into account that there are no concrete transitions from states in S_{k-1}^{reach} to states in S_k. We therefore require that the set $\hat{E}'$ of A' does **not** contain transitions in the set $\{(\alpha'(s_1), \alpha'(s_2)) \mid \exists\, s_1 \in S_{k-1}^{reach}, s_2 \in S_k\}$. Thus, succeeding refined models will exclude previously explored counterexamples. A method for the refinement of abstract models for infinite-state systems will be presented in the next section.

3 Refinement of Abstract Models for Infinite State Systems

This section presents a specific method for refining an abstract model A for an infinite state system. The main idea is to directly use the information obtained from the validation procedure to refine some abstract states: Assume that the abstract model includes a transition between $\hat{s}_1$ and $\hat{s}_2$, while the validation of the counterexample has revealed that only a subset of concrete states in $S_2 := \alpha^{-1}(\hat{s}_2)$ is reachable from concrete states in $S_1 := \alpha^{-1}(\hat{s}_1)$. In this case we refine A by splitting $\hat{s}_2$ into two new states. The first one, denoted by $\hat{s}_2^{reach}$, represents the reachable subset of S_2, given by $S_2^{reach} := succ(S_1) \cap S_2$. The second one, denoted by $\hat{s}_2^{comp}$, represents the complement of the reachable part, given by $S_2^{comp} := S_2 \setminus S_2^{reach}$. In addition, the abstraction function that maps concrete states to abstract ones has to be refined, too.

Definition 6 *Refinement by State Splitting.* Given a concrete model $C = (S, S_0, E)$ and an abstract model $A = (\hat{S}, \hat{S}_0, \hat{E})$ with an abstraction function $\alpha : S \to \hat{S}$. Let $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ be a transition of a counterexample $\hat{\sigma}$. Then, we define ρ_{split} as a refinement function that maps A, α, and $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ onto the refined abstract model $A' = (\hat{S}', \hat{S}'_0, \hat{E}')$ and the refined abstraction function $\alpha' : S \to \hat{S}'$, i.e., $(A', \alpha') = \rho_{split}(A, \alpha, (\hat{s}_1, \hat{s}_2))$, defined as follows:

- $\hat{S}' = (\hat{S} \setminus \hat{s}_2) \cup \{\hat{s}_2^{reached}, \hat{s}_2^{comp}\}$
- $\alpha'(s) = \begin{cases} \alpha(s) & \text{if } s \notin S_2 \\ \hat{s}_2^{reach} & \text{if } s \in S_2^{reach} \\ \hat{s}_2^{comp} & \text{if } s \in S_2^{comp} \end{cases}$
- $\hat{S}'_0 = \{\hat{s}' \in \hat{S}' \mid \alpha''(\hat{s}') \in \hat{S}_0\}$
- $\hat{E}' = \{(\hat{s}'_1, \hat{s}'_2) \in \hat{S}' \times \hat{S}' \mid \exists \hat{s}_1, \hat{s}_2 \in \hat{S}\ : (\hat{s}_1, \hat{s}_2) \in \hat{E} \wedge \hat{s}_1 = \alpha''(\hat{s}'_1) \wedge \hat{s}_2 = \alpha''(\hat{s}'_2)\} \setminus (\hat{s}_1, \hat{s}_2^{comp})$

where $\alpha'' : \hat{S}' \to \hat{S}$ maps $\hat{s}'$ onto itself if $\hat{s}' \notin \{\hat{s}_2^{reached}, \hat{s}_2^{comp}\}$, and on $\hat{s}_2$ otherwise. $\diamond$

Lemma 2. *Let $A = (\hat{S}, \hat{S}_0, \hat{E})$ be an abstract model of $C = (S, S_0, E)$ with the abstraction function $\alpha : S \to \hat{S}$. For a given transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$, assume that $S_2^{reach} \neq \emptyset$ holds. Then, $(A', \alpha') := \rho_{split}(A, \alpha, (\hat{s}_1, \hat{s}_2))$ satisfies $A \succeq A' \succeq C$.* $\square$

As a next step, we consider the case where the set of successors of S_1 and the set S_2 are disjoint. In this case, we can simply omit the corresponding abstract transition.

Definition 7 *Refinement by Eliminating a Transition.* The function ρ_{purge} is a refinement that maps an abstract model $A = (\hat{S}, \hat{S}_0, \hat{E})$, an abstraction function $\alpha : S \to \hat{S}$ and a transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ onto $A' = (\hat{S}, \hat{S}_0, \hat{E}')$ with $\hat{E}' = \hat{E} \setminus (\hat{s}_1, \hat{s}_2)$. $\diamond$

Lemma 3. *Let $A = (\hat{S}, \hat{S}_0, \hat{E})$ be an abstract model of $C = (S, S_0, E)$ with the abstraction function $\alpha : S \to \hat{S}$. For a given transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$, assume that $S_2^{reach} = \emptyset$ holds. Then, $A' := \rho_{purge}(A, \alpha, (\hat{s}_1, \hat{s}_2))$ satisfies $A \succeq A' \succeq C$.* $\square$

Based on these results, we now present a more specific formulation of the CEGAR algorithm in Fig. 2, called INFINITE-STATE-CEGAR, which uses the functions ρ_{split} and ρ_{purge} for refinement.

```
ALGORITHM: INFINITE-STATE-CEGAR
INPUT: Concrete model $C$ and a set of bad states $B$
OUTPUT: $B$ is (or is not) reachable

Generate initial abstract model $A$ and abstraction function $\alpha$
$\hat{B} := \alpha(B)$
Generate counterexample $\hat{\sigma} = (\hat{s}_0, \ldots, \hat{s}_m)$ by model checking of $A$ wrt. $\hat{B}$
$S_0^{reach} := \alpha^{-1}(\hat{s}_0)$
WHILE $\hat{\sigma}$ exists DO
    // validation of counterexample
    $k := 0$
    WHILE $S_k^{reach} \neq \emptyset$ AND $k < m$ DO
        $k := k + 1$
        $S_k^{reach} := succ(S_{k-1}^{reach}) \cap \alpha^{-1}(\hat{s}_k)$
    ENDDO
    // if counterexample is validated, then terminate, else refine
    IF $S_k^{reach} \neq \emptyset$ THEN terminate with "B reachable"
    ELSE
        FOR $l = 1, \ldots, k-1$
            // split abstract state $\hat{s}_l$ into two: one that corresponds
            // to $S_l^{reach}$ and one that corresponds to $\alpha^{-1}(\hat{s}_l) \setminus S_l^{reach}$
            IF $S_l^{reach} \neq \alpha^{-1}(\hat{s}_l)$
            THEN $(A, \alpha) := \rho_{split}(A, \alpha, \hat{s}_{l-1}, \hat{s}_l)$
            ENDIF
        ENDFOR
        // remove spurious transition between $\hat{s}_{k-1}$ and $\hat{s}_k$
        $A := \rho_{purge}(A, \alpha, \hat{s}_{k-1}, \hat{s}_k)$
        Generate $\hat{\sigma}$ by model checking of $A$ wrt. $\hat{B}$
    ENDIF
ENDDO
Terminate with "B not reachable"
```

Fig. 2. INFINITE-STATE-CEGAR.

Correctness of the algorithm is implied by the following two lemmas.[1] Note that termination of the algorithm cannot be guaranteed as the number of states in the concrete model may be infinite, and a finite abstract model to verify (or disprove) the given property may not exist.

[1] The proofs of all lemmas in the paper can be found in the Appendix.

Lemma 4. *If the algorithm terminates with "B reachable", then* $C \not\models$ ***AG***$\neg B$. □

Lemma 5. *If the algorithm terminates with "B not reachable", then* $C \models$ ***AG***$\neg B$. □

The proposed procedure of validating counterexamples and refining abstract models is based on the computation of successor states. Alternatively, one could formulate a similar algorithm that uses sets of predecessors, or even a combination of both as presented in [1] and [10].

The INFINITE-STATE-CEGAR algorithm in Fig. 2 is based on the assumption that sets of successor states are exactly computable. Lemma 5 holds, however, also if successor states are not exactly computable, and instead only *over*-approximations of the set of successor states can be computed. If only under-approximations of successor sets can be computed, Lemma 5 will not hold, but Lemma 4 will. For the class of hybrid systems considered in the following section only over-approximations of successor sets are computable.

4 Hybrid Systems

Hybrid systems are a class of infinite state systems that include both continuous and discrete state variables. This section presents the syntax and semantics of hybrid automata, which are used to model hybrid systems. We will illustrate these definitions with an example that models a simple car controller. The same example will be used in later sections to illustrate our new approach to the verification of hybrid systems.

4.1 Definition of Hybrid Automata

Definition 8 *Syntax of the Hybrid Automaton* HA. A *hybrid automaton* is a tuple $HA = (Z, z_0, X, inv, X_0, T, g, j, f)$ where

- Z is a finite set of *locations* with an *initial location* $z_0 \in Z$.
- $X \subseteq \mathbb{R}^n$ is the continuous state space.
- $inv : Z \rightarrow 2^X$ assigns to each location $z \in Z$ an invariant of the form $inv(z) \subseteq X$.
- $X_0 \subseteq X$ is the set of initial continuous states. The set of initial hybrid states of *HA* is thus given by the set of states $\{z_0\} \times X_0$.
- $T \subseteq Z \times Z$ is the set of *discrete transitions* between locations.
- $g : T \rightarrow 2^X$ assigns a *guard* set $g((z_1, z_2)) \subseteq X$ to $t = (z_1, z_2) \in T$.
- $j : T \times X \rightarrow 2^X$ assigns to each pair $(z_1, z_2) \in T$ and $x \in g((z_1, z_2))$ a *jump* set $j((z_1, z_2), x) \subseteq X$.
- $f : Z \rightarrow (X \rightarrow \mathbb{R}^n)$ assigns to each location $z \in Z$ a continuous vector field $f(z)$. We use the notation f_z for $f(z)$. The evolution of the continuous behavior in location z is governed by the differential equation $\dot{\chi}(t) = f_z(\chi(t))$. We assume that the differential equation has a unique solution for each initial value $\chi(0) \in X_0$. ⋄

The semantics of HA is defined by means of a trace transition system. Each state (z, x) in the trace transition system corresponds to a continuous state x within location z. Two such states, (z_1, x_1) and (z_2, x_2), are connected by a transition in the trace transition system if and only if state (z_2, x_2) can be reached from state (z_1, x_1) by a continuous evolution within location z_1 followed by a discrete transition to location z_2.

Definition 9 *Semantics of the Hybrid Automaton* HA. The semantics of a Hybrid automaton HA is a *transition system* $TTS = (S, S_0, E)$ with:

– the set of all *hybrid states* (z, x) of HA,

$$S = \bigcup_{z \in Z} \bigcup_{x \in inv(z)} (z, x) \tag{1}$$

– the set of *initial hybrid states* $S_0 = \{z_0\} \times X_0$,
– transitions $(s_1, s_2) \in E$ with $s_1 = (z_1, x_1)$, $s_2 = (z_2, x_2)$, iff there exists $(z_1, z_2) \in T$ and a trajectory $\chi : [0, \tau] \rightarrow X$ for some $\tau \in \mathbb{R}^{>0}$ such that:
  - $x_1 = \chi(0)$, $\chi(\tau) \in g((z_1, z_2))$,
  - $x_2 \in j((z_1, z_2), \chi(\tau))$,
  - $\dot{\chi}(t) = f_{z_1}(\chi(t))$ for $t \in [0, \tau]$,
  - $\chi(t) \in inv(z_1)$ for $t \in [0, \tau]$,
  - $x_2 \in inv(z_2)$.

A path $\sigma = \{s_0, s_1, s_2, \ldots\}$ of TTS is called a *trace* of HA, and we refer to TTS as the *trace transition system* of HA. $\diamond$

Definition 10 *Safety of a Hybrid Automaton.* For a hybrid automaton HA with a semantics as in Defn. 9, let $z_b \in Z \setminus \{s_0\}$ denote an *unsafe* location. HA is said to be *safe* with respect to z_b, denoted by $TTS \models \mathbf{AG} \neg z_b$ iff for all traces σ applies: $\nexists s \in \sigma$ with $s = (z_b, x)$ for some $x \in X$. We write $TTS \not\models \mathbf{AG} \neg z_b$ otherwise. $\diamond$

The extension of the analysis task to multiple initial locations and/or multiple unsafe locations is straightforward but is omitted here for simplicity.

4.2 Example

As a motivating example, we use a simple controller that steers a car along a straight road. The car is assumed to drive at a constant speed $r = 2$, and its motion is modeled by the horizontal position x ($x = 0$ corresponds to the middle of the road) from the middle of the road and the heading angle γ ($\gamma = 0$ corresponds to moving in the vertical direction). Fig. 3 shows a scenario in which the car drives initially on the road. The controller is able to detect whether the car is on the left or right border (i.e. $x \leq -1$, $x \geq 1$) – whenever the car enters the left border, the controller forces it to turn right until the car is back on the road again. Then a left turn is initiated, and continued until the car is again going straight ahead in the direction of the road, i.e. when the heading is aligned with the road ($\gamma = 0$). A similar strategy is employed when the car enters the right border.

Fig. 4 shows a hybrid automaton model of the controlled behavior for the car. Besides the position x and the heading angle γ, the description includes an internal timer c, that the controller uses to time the steering manoeuvres. The differential equations for these three continous variables depend on the location: we have $\dot{x} = -r \cdot sin(\gamma)$ in all locations except of `in_canal`. The derivative of γ varies when a border is reached. On the border the motion of the car describes an arc with the angular velocity $\dot{\gamma} = -\omega = -\pi/4$ (or $\omega = \pi/4$ respectively), i. e., the arc is part of a circle with radius r/ω. The timer c measures the time period which the car spends on a borders. In the correction modes the timer decreases with double rate, i.e., the correction takes half the time as the car was on the border before. Since the sign of $\dot{\gamma}$ is reversed when the car moves back on the road, the angle has the value zero when the correction mode is left ($c = 0$), i.e., the car moves then along the road. During this correction it might, however, happen that the

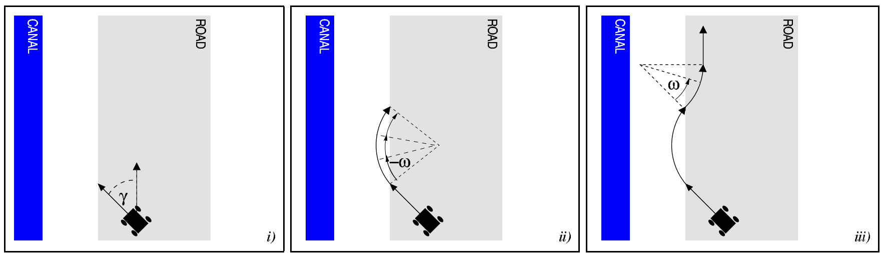


Fig. 3. *i)* Initially, the car drives on the road with heading angle γ. *ii)* If the controller detects that the car left the road, it corrects the heading by turning right to avoid the canal. *iii)* Once the car is back on the road, a left turn is initiated until the car moves straight again.

other border is reached, which means that the controller then switches to the strategy of the corresponding location.

The three continuous variables are initialized to $-1 \leq x \leq 1$ (the car is on the road), $-\pi/4 \leq \gamma \leq \pi/4$, and $c = 0$. It has to be verified for this set of initial states whether the given control strategy guarantees that the unsafe location `in_canal` (z_b) is never reached. The following sections present how this task can be solved by abstraction-based and counterexample-guided verification.

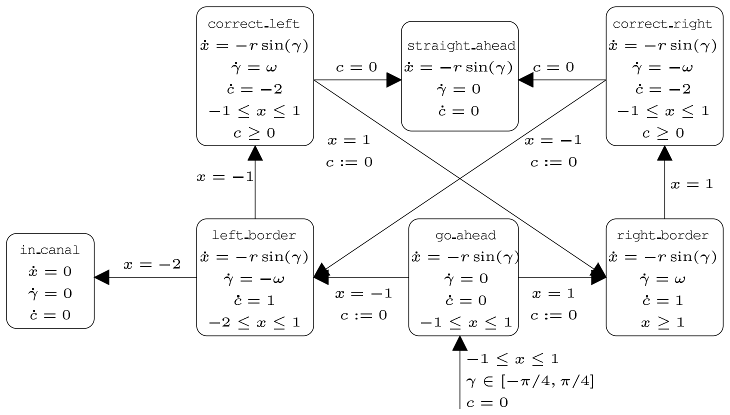


Fig. 4. Hybrid automaton that models the car steering example. Location `in_canal` has to be avoided. For each location, the continuous dynamics of the three variables x, γ and c is described by differential equations, and invariants are specified as inequalities. Guards and jumps are assigned to the transitions, e.g., a transition from location `go_ahead` to `left_boarder` is possible if the value of x is 1, and then the value of c is set to zero.

5 Refinement of Abstractions for Hybrid Systems

This section applies the general concepts of Section 3 to the particular class of infinite state systems of hybrid systems.

We present specific solutions for the two crucial steps, the validation of counterexamples and the refinement of abstract models. The key to the validation step is the computation of successor states for a given set of states in the trace transition system. Starting from the initial set, the validation procedure computes the successors along the counterexample until either the unsafe location z_{sp} is reached or a transition is determined to be spurious. The computation of sets of successors states is usually the most expensive step in hybrid system verification. Moreover, successor sets can be computed and represented *exactly* only for certain sub-classes of hybrid systems [15, 16]. However, several approaches to over-approximate successor sets have been published, as e. g., successor set approximations by orthogonal polyhedra [17], general polyhedra [18], projections to lower dimensional polyhedra [19], or ellipsoids [20]. Most of these approaches aim at providing an efficient way to obtain conservative but tight approximations to sets of reachable states for hybrid systems.

The verification framework presented here can include different techniques to over-approximate the set of successors. The idea of using different methods is motivated by the trade-off between the accuracy and the computational complexity of different methods. If, e.g., a faster but maybe less accurate technique is sufficient to refute a counterexample, there is no need to use a more computationally expensive method.

In the following, we first describe how an initial abstraction for a hybrid automaton can be obtained, and then focus on the validation of counterexamples and the refinement based on the use of different methods for computing successor states.

5.1 Abstraction of Hybrid Systems

For the first step of the INFINITE-STATE-CEGAR algorithm, the construction of an initial abstraction, we introduce one abstract state for each location of HA. This means that two hybrid states (z_i, x_i) and (z_j, x_j) of TTS are mapped to the same abstract state if and only if $z_i = z_j$. This rule applies for all but the initial location, for which we introduce one abstract state $\hat{s}_0$ to represent all initial hybrid states of TTS, and another one ($\hat{s}_0'$) to represent the remaining hybrid states corresponding to the location z_0:

Definition 11 *Initial Abstraction of Hybrid Systems.* Given a hybrid automaton HA with $Z = \{z_0, z_1, \ldots, z_{n_z}\}$, let S denote the set of hybrid states as defined in (1). For $i \in \{0, 1, \ldots, n_z\}$, we define the abstraction function $\alpha : S \to \hat{S}$ by:

$$\alpha(z_i, x) = \begin{cases} \hat{s}_0 \text{ if } i = 0 \wedge x \in X_0 \\ \hat{s}_0' \text{ if } i = 0 \wedge x \notin X_0 \\ \hat{s}_i \text{ otherwise} \end{cases} \tag{2}$$

and the initial abstract model $A = (\hat{S}, \hat{S}_0, \hat{E})$ is defined by ($i \in \{0, 1, \ldots, n\}$, $j \in \{0, 1, \ldots, n_z\}$):

- $\hat{S} = \{\hat{s}_0', \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n\}$
- $\hat{S}_0 = \{\hat{s}_0\}$
- $\hat{E} = \{(\hat{s}_i, \hat{s}_j) | (z_i, z_j) \in T\} \cup \{(\hat{s}_0', \hat{s}_j) | (z_0, z_j) \in T\} \cup \{(\hat{s}_i, \hat{s}_0') | (z_i, z_0) \in T\}$ $\diamond$

The initial abstract model represents the discrete structure of the hybrid system without regarding the continuous dynamics and guards. Given this definition, it has to be shown that A is indeed an abstract model of the underlying trace transition system, i.e., that it fulfills Defn. 2:

Lemma 6. *For HA with trace transition system $TTS = (S, S_0, E)$, let $A = (\hat{S}, \hat{S}_0, \hat{E})$ denote the initial abstract model for TTS. Then, $A \succeq TTS$.* $\square$

Example (cont.) Fig. 5 depicts the initial abstract model of the hybrid system in Fig. 4. It is a copy of the discrete part of the hybrid system, except that the initial location is divided into two parts: $\hat{s}_0$ represents the states in location `go_ahead` with $x \in [-1,1]$, $\gamma \in [-\pi/4, \pi/4]$ and $c = 0$, and $\hat{s}_0'$ all other states in `go_ahead`. The abstract states $\hat{s}_1$ to $\hat{s}_6$ represent the hybrid states of the other locations (`left_border`, `right_border`, `correct_left`, `correct_right`, `straight_ahead` and `in_canal`, respectively). ⧫

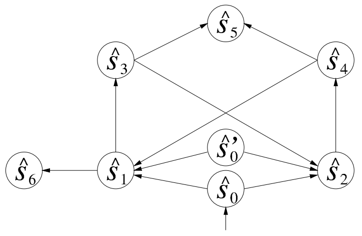


Fig 5. Initial abstract model of the hybrid system depicted in Fig. 4

5.2 Over-approximation of the Sets of Successors

We now turn to the point of computing sets of successor states, as required in the validation and refinement steps. The goal is to use different over-approximations with different precisions and different computational needs. We first define an over-approximation operator of the successor relation for a tuple of sets of states. The operator conservatively approximates which states in the second set (target set) are successors of states in the first set (source set).

Definition 12 *Over-approximation of successor states.* Let HA be a hybrid automaton with the trace transition system $TTS = (S, S_0, E)$, and let A and α be defined as in Defn. 11. For a transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ of A, we call $S_1 := \alpha^{-1}(\hat{s}_1)$ the set of *hybrid source states* and $S_2 := \alpha^{-1}(\hat{s}_2)$ the set of *potential hybrid successor states*. Then, $\overline{succ} : (2^S \times 2^S) \to 2^S$ is an *over-approximation* of the hybrid successor states in S_2 iff the following holds:

- $\overline{succ}(S_1, S_2) \subseteq S_2$,
- for all $s_1 \in S_1$ and $s_2 \in S_2 \setminus \overline{succ}(S_1, S_2)$, $(s_1, s_2) \notin E$. ◇

A possible explicit realization of the operator $\overline{succ}$ combines the following steps: (a) By approximating the continuous evolution for all states in S_1, the reachable subset of the guard set $g(t)$ is determined, where $t = (z_1, z_2) \in T$ is the transition of HA that corresponds to the transition $(\hat{s}_1, \hat{s}_2) \in \hat{E}$ of A. Usually, this step is the most costly of the whole verification procedure; (b) the jump function $j(t, x)$ is applied to all hybrid states (z_1, x) which are in the reachable subset of $g(t)$; (c) the image of $j(t, x)$ is intersected with the set S_2 of potential hybrid successor states.

Example (cont.) Our prototype implementation uses two different methods, $\overline{succ}_{coarse}$ and $\overline{succ}_{tight}$, to over-approximate the set of successor states. Fig. 6 illustrates these two methods for the discrete transition from `correct_right` to `left_border`. For location `correct_right` we choose S_1 as subset of the plane $x = 1$, and S_2 as all states of location `left_border` that satisfy the invariant $-2 \leq x \leq -1$. Fig. 6 depicts S_1 and the face of S_2 that coincides with the guard $x = -1$. The transition is not spurious, if there exists a trajectory that starts in S_1, and ends in S_2 without leaving the invariant of `correct_right` ($-1 \leq x \leq 1 \wedge c \geq 0$). Fig 6 i) depicts a number of trajectories that start in S_1, none of them reaches S_2.

The first method $\overline{succ}_{coarse}$ poses the existence question for a trajectory between S_1 and S_2 as an optimization problem. The distance between a trajectory and S_2 is defined as the minimum distance between all points on the trajectory and S_2. If the

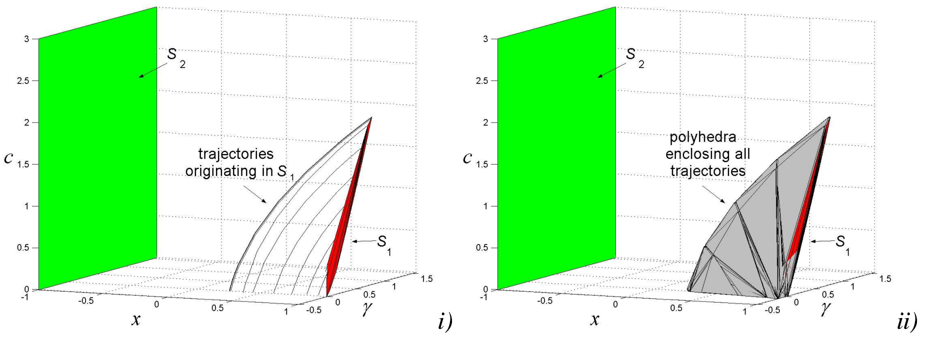

S 2
trajectories originating in S 1
S 1
c
x
γ
i)
polyhedra enclosing all trajectories
ii)

Fig. 6. All trajectories that originate in S_1 leave the invariant when $c = 0$, and none of them comes close to S_2. Figure *i)* shows the result of the optimization method. Figure *ii)* the result of the method that enclose the trajectories by polyhedra.

global minimum over all trajectories that start in S_1 is strictly greater than zero, then no successor state of S_1 exists in S_2. In this case $\overline{succ}_{coarse}$ returns an empty set. If the minimum distance is zero, at least one corresponding concrete path exists, and $\overline{succ}_{coarse}$ returns the complete set S_2 as an over-approximation of the set of successor states. The bold trajectory in Fig. 6 i) is the optimal trajectory. Its distance to S_2 is greater than zero, and there is hence no trajectory from S_1 to S_2.

The second method $\overline{succ}_{tight}$ computes polyhedra that encloses all trajectories that originate in S_1. This over-approximation with polyhedra is based on work presented in [18]. The set of successor states $\overline{succ}_{tight}(S_1, S_2)$ is then obtained by intersecting the polyhedra with S_2. Fig. 6 ii) shows that this intersection is empty, i.e. there are no successors of S_1 in S_2. ♦

5.3 Validation and Refinement

The INFINITE-STATE-CEGAR algorithm makes a clear distinction between the validation of a counterexample, and the refinement of the abstract model. For hybrid systems, we propose a slightly different approach, in which the steps of validation and refinement are interleaved. We assume to have a set of over-approximation techniques $\overline{succ}_1, \ldots, \overline{succ}_n$ that can (but not necessarily need to) establish a hierarchy of coarse to tight approximations.

The proposed algorithm for the combined validation and refinement steps of a counterexample is shown in Fig. 7. Let $\sigma = (\hat{s}_0, \ldots, \hat{s}_m)$ denote a counterexample of the abstract model A. The algorithm consists of two nested loops. The outer loop corresponds to checking each transition of the counterexample. The inner loop applies each of the over-approximation techniques to the current transition of the counterexample, and, depending on the result, one of the two refinement operations is executed: If an over-approximation technique $\overline{succ}_l$ reveals that the current transition is spurious, i.e. $S_k^{reach} = \emptyset$, then the transition is removed from the abstract model by ρ_{purge}. When a transition is removed, the set of behaviors of A does not include the current counterexample anymore, and thus the combined validation and refinement of the current counterexample is completed.

If on the other hand, $\overline{succ}_l$ returns a non-empty set S_{reach}^k and this set is a true subset of the states corresponding to $\hat{s}_k$, the function ρ_{split} divides $\hat{s}_k$ into two states $\hat{s}_k^{reach}$ and $\hat{s}_k^{comp}$ (cf. Defn. 6). In this case however $\sigma = (\hat{s}_0, \ldots, \ldots, \hat{s}_{k-1}, \hat{s}_k^{reach}, \hat{s}_{k+1} \ldots, \hat{s}_m)$

```
FOR $k = 1, \ldots, m$
    FOR $l = 1, \ldots, n$
        $S_k^{reach} := \overline{succ}_l(S_{k-1}^{reach}, \alpha^{-1}(\hat{s}_k))$
        IF $S_k^{reach} = \emptyset$
            $A := \rho_{purge}(A, \hat{s}_{k-1}, \hat{s}_k)$,
            RETURN //jump out of both loops
        ELSEIF $S_k^{reach} \subsetneq \alpha^{-1}(\hat{s}_k)$
            $(A, \alpha) := \rho_{split}(A, \alpha, \hat{s}_{k-1}, \hat{s}_k, S_k^{reach})$
        ENDIF
    ENDFOR
ENDFOR
```

Fig. 7. Refinement and Validation Steps for Hybrid Systems.

remains a counterexample of the refined model. Thus the algorithm continues with the next transition $(k+1)$ until either $S_k^{reach} = \emptyset$ or until the last transition of the counterexample is validated.

There is some freedom in combining the steps of validation and refinement, i. e., the scheme in Fig. 7 is just one possible implementation. One interesting alternative is to apply the coarsest method for validation first to all transitions in the abstract counterexample, or to apply state splitting (ρ_{split}) only based on the result of the most accurate approximation method $\overline{succ}_n$.

The algorithm as proposed in Fig. 7 has two possible outcomes: either it is proved that a forbidden state cannot be reached or that there exists a counterexample that cannot be refuted. Since the validation procedure relies on over-approximations, it can not be guaranteed that this abstract counterexample corresponds to a concrete one. In this case, under-approximations of sets of successor states can possibly be used to prove that a counterexample exists: Assume that the procedure terminates with a counterexample $\sigma = (\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_k, \ldots, s_m)$, no transition of which could be refuted. Similar to Defn. 12, we can define an *under-approximation* of successor states $S_k^{reach} = \underline{succ}(S_{k-1}^{reach}, \alpha^{-1}(\hat{s}_k))$ which returns a set $S_k^{reach} \subseteq \alpha^{-1}(\hat{s}_k)$ for which it is ensured that it only contains true successors of S_{k-1}^{reach}. If this operator is applied along the counterexample (from $k = 1$ to $k = m$) and $S_n^{reach} \neq \emptyset$ applies, there exists at least one path for the hybrid system which violates the safety property.

Example (cont) The requirement that the hybrid model in Fig. 4 should never enter the location `in_canal` translates into the reachability question for state $\hat{s}_6$ of the abstract model in Fig. 5. The first counterexample for the initial abstract model is $\sigma_1 = (\hat{s}_0, \hat{s}_1, \hat{s}_6)$ (see Fig. 8(i)). The validation procedure considers first the transition $(\hat{s}_0, \hat{s}_1)$ which corresponds to the transition between `go_ahead` and `left_border` in the hybrid automaton. As a first step, $\overline{succ}_{coarse}(S_0, \alpha^{-1}(\hat{s}_1))$ is computed with the result that the minimum distance over all initial states is zero. This is obvious from the fact that those states of the initial set for which $x = -1$ enable the transition guard immediately. Thus, $\overline{succ}_{coarse}$ returns the entire invariant of location `left_border` as set S_2. The next step is to compute $S_2^{reach} = \overline{succ}_{tight}(S_0, \alpha^{-1}(\hat{s}_1))$. The algorithm then splits $\hat{s}_1$ such that $\hat{s}_1$ represents the set S_2^{reach}, and the new abstract state $\hat{s}_1'$ represents $S_2 \setminus S_2^{reach}$ (Fig. 8 (ii)).

Since the counterexample has not been eliminated yet, the transition $(\hat{s}_1, \hat{s}_6)$ is considered next. Method $\overline{succ}_{coarse}$ finds that the minimal distance between the trajectories that start in S_2^{reach}, and the guard $x = -2$ is greater than zero. This means no trajectory reaches the guard, and the corresponding transition is removed (Fig. 8 (iii)).

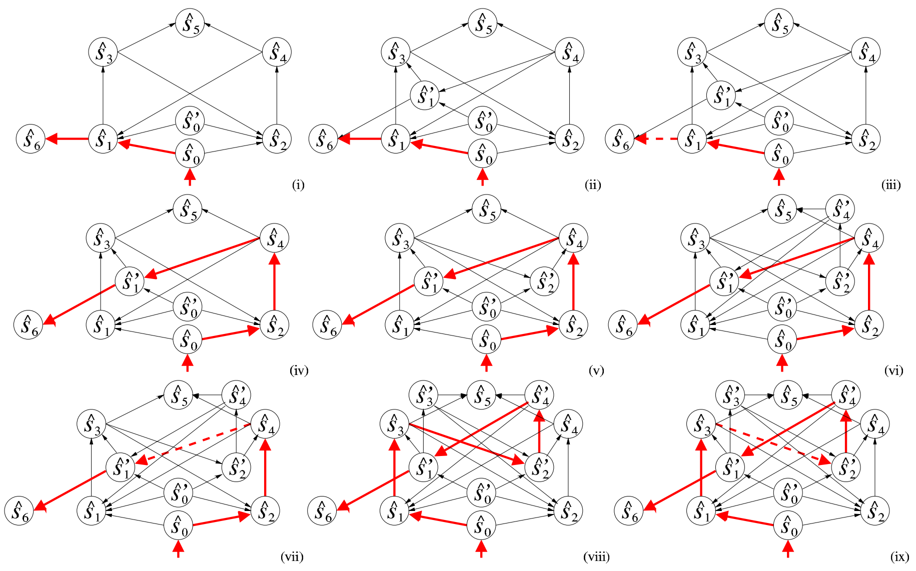


Fig. 8. Counterexample guided abstraction illustrated for the car steering problem.

The procedure continues with the next counterexample $\sigma_2 = (\hat{s}_0, \hat{s}_2, \hat{s}_4, \hat{s}'_1, \hat{s}_6)$, as depicted in Fig. 8 (iv). As for the first counterexample, the abstract state $\hat{s}_2$ is split into the states that are reachable from the initial set S_0, and the remainder (Fig. 8 (v)). Then, the procedure moves one transition ahead and splits state $\hat{s}_4$ as a result of applying $\overline{succ}_{tight}$. The reachable part is represented by $\hat{s}_4$ in Fig. 8 (vi). Method $\overline{succ}_{coarse}$ then finds that one cannot reach any state that is represented by $\hat{s}'_1$ from this set, and the transition $(\hat{s}_4, \hat{s}'_1)$ can be deleted from A (Fig. 8 (vii)).

The final counterexample is $\sigma_3 = (\hat{s}_0, \hat{s}_1, \hat{s}_3, \hat{s}'_2, \hat{s}'_4, \hat{s}'_1, \hat{s}_6)$. The state $\hat{s}_1$ was already split for the first counterexample. Similarly to the procedure for the counterexample σ_2, abstract state $\hat{s}_3$ is split as depicted in Fig. 8 (viii). It can then be shown that transition $(\hat{s}_3, \hat{s}'_2)$ is spurious, which eliminates the last counterexample (Fig. 8 (ix)). Consequently, the abstract state $\hat{s}_6$ is not reachable, and thus the same applies for the location `in_canal` of the hybrid automaton. ♦

5.4 Experimental Results

Experimental results for a prototype implementation of the procedure indicate its advantages over existing methods. We compare INFINITE-STATE-CEGAR with a method based on breadth-first application of the successor operator $\overline{succ}_{tight}$. Breadth-first application is the most prevalent method used for model checking hybrid systems. This approach needs 175 second cputime on a Pentium 4, 1.4GHz, to compute that location `in_canal` is not reachable.

INFINITE-STATE-CEGAR together with only one of the two over-approximation methods, $\overline{succ}_{tight}$, takes about 120 seconds to verify that the system satisfies the property. As in in the case of the breadth-first methods, 99% of the cputime is spend on computing $\overline{succ}_{tight}$. If INFINITE-STATE-CEGAR employs both approximation methods, then the time is cut in about half. The algorithm takes 68 seconds for the verification, of which 64 seconds ares used to compute $\overline{succ}_{tight}$, and 3 seconds to solve the optimization problems of $\overline{succ}_{coarse}$.

6 Conclusions

This paper presents a new method for using counterexamples to refine abstractions of hybrid systems. The principal alternative for verifying the safety properties considered in this paper is to compute the reachable states for the hybrid system using a breadth-first application of the successor operator $succ$. It is apparent that the INFINITE-STATE-CEGAR procedure can be faster than breadth-first reachability when the safety property does not hold for the concrete system, since in this case it is possible that the model checker will quickly find a true counterexample. On the other hand, if the safety property holds, refuting one counterexample may implicitly refute others. However, the INFINITE-STATE-CEGAR procedure may continue until all possible counterexamples have been explored (and indeed, may not terminate), which is in some cases equivalent to the breadth-first reachability computation. Nevertheless, INFINITE-STATE-CEGAR offers the possibility of using multiple methods for computing approximations to the successor states. Further evaluation of the INFINITE-STATE-CEGAR procedure and a comparison of INFINITE-STATE-CEGAR to breadth-first reachability as well as other alternatives is currently underway.

A Proofs

Proof of Lemma 1.

Proof. By contradiction: If $C \not\models$ **AG**$\neg B$, then at least one path $\sigma = (s_0, s_1, \ldots, b)$ with $b \in B$ must exist for C. From Defn. 2, it follows that the corresponding abstract counterexample $\hat{\sigma} = (\hat{s}_0, \hat{s}_1, \ldots, \hat{b})$ of A is a counterexample which contradicts the premise $A \models$ **AG**$\neg\hat{B}$. ■

Proof of Lemma 2.

Proof. (i) $A \succeq A'$. It follows straightforwardly that A is an abstract model of A' with abstraction function α'' as defined in Defn. 6.
(ii) $A' \succeq C$. From the above definitions of $A' = (\hat{S}', \hat{S}'_0, \hat{E}')$ and α', it follows that A' would be an abstract model of C, if $\hat{E}'$ also included the transition $(\hat{s}_1, \hat{s}_2^{comp})$. However, since S_2^{reach} and S_2^{comp} are disjoint, this abstract transition does not correspond to any concrete transition and can therefore be omitted. ■

Proof of Lemma 3.

Proof. (i) $A \succeq A'$. The corresponding abstraction function is the identity. Since A has just an additional transition it is an abstract model of A'.
(ii) $A' \succeq C$. The abstraction function for this abstraction is α. We can then omit the abstract transition $(\hat{s}_1, \hat{s}_2)$, since it does not correspond to any concrete transition. ■

Proof of Lemma 4.

Proof. If the algorithm terminates with "B reachable", then the set of reachable states in the concrete model is non-empty along the path of the last checked counterexample. Formally, $S_k^{reach} \neq \emptyset$, $k = 0, \ldots, m$ due to the conditions in the IF statement ($S_k^{reach} \neq \emptyset$) and the WHILE statement ($S_k^{reach} \neq \emptyset$ AND $k < m$).

We can now show that the last checked counterexample in the algorithm is not spurious. To do so, we first show that for each k, all $s_k \in S_k^{reach}$ can be reached by paths in the concrete model. The proof is done by induction on k. For $k = 0$, each

$s_0 \in S_0^{reach}$ can be reached by a path of length zero. For $k > 0$, for each $s_k \in S_k^{reach}$ there exists an $s_{k-1} \in S_{k-1}^{reach}$ such that $(s_{k-1}, s_k) \in E$ (by definition of the succ operator). By induction, s_{k-1} is reachable by some concrete path $(s_0, \ldots, s_{k-1})$, hence s_k is reachable via the concrete path $(s_0, \ldots, s_k)$.

Since for each k, all $s_k \in S_k^{reach}$ can be reached by paths in the concrete model, there are paths $(s_0, s_1, \ldots, s_m)$ with $s_m \in S_m^{reach}$. Each such path corresponds to a counterexample in the concrete model, i.e. $S_m^{reach} \subseteq B$, since $\alpha(s_m) \in \hat{B}$ (as the path is a counterexample in the abstract model), and $\alpha(s_m) \in \hat{B}$ implies $s_m \in B$. Thus, $C \models \mathbf{AG}\neg B$ ■

Proof of Lemma 5.

Proof. The algorithm terminates only if it was not possible to find any counterexample for the current abstract model A. But since A is in each step an abstraction of C we can conclude by Lemma 1 that $C \models \mathbf{AG}\neg B$ holds. ■

Proof of Lemma 6.

Proof. We show that α as defined in Def. 11 is an abstraction function. The first condition in Def. 2 follows directly from the definition of α. To show the second condition, it must be proved that

$\hat{E} = \{(\hat{s}_i, \hat{s}_j) | (z_i, z_j) \in T\} \cup \{(\hat{s}'_0, \hat{s}_j) | (z_0, z_j) \in T\} \cup \{(\hat{s}_i, \hat{s}'_0) | (z_i, z_0) \in T\} \supseteq \{(\hat{s}_i, \hat{s}_j) | \exists s_i, s_j \in S \; : (s_i, s_j) \in E, \hat{s}_i = \alpha(s_i), \hat{s}_j = \alpha(s_j)\}$.

Assume $(s_i, s_j) \in E$, and $s_i = (z_i, x_i)$ and $s_j = (z_j, x_j)$ with $x_i, x_j \in X$ and $i, j \neq 0$. Then, it follows from the definition of E in Def. 9 that $(z_i, z_j) \in T$. Thus, $(\hat{s}_i, \hat{s}_j) \in \hat{E}$. The other cases ($i = 0$ or $j = 0$) can be shown in a similar way. ■

References

1. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. Volume 1855 of LNCS., Springer (2000) 154–169
2. Kurshan, R.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press (1994)
3. Silva, B., Stursberg, O., Krogh, B., Engell, S.: An assessment of the current status of algorithmic approaches to the verification of hybrid systems. In: IEEE Conf. on Decision and Control. (2001) 2867–2874
4. Henzinger, T., Minea, M., Prabhu, V.: Assume-guarantee reasoning for hierarchical hybrid systems. In: HSCC. Volume 2034 of LNCS., Springer (2001) 275–290
5. Frehse, G., Stursberg, O., Engell, S., Huuck, R., Lukoschus, B.: Modular analysis of discrete controllers for distributed hybrid systems. In: IFAC World Congress. (2002)
6. Alur, R., Henzinger, T., Lafferriere, G., Pappas, G.: Discrete abstractions of hybrid systems. Proceedings of the IEEE **88** (2000) 971–984
7. Alur, R., Dang, T., Ivancic, F.: Reachability analysis of hybrid systems via predicate abstraction. In: HSCC. Volume 2289 of LNCS., Springer (2002) 35–48
8. Tiwari, A., Khanna, G.: Series of abstractions for hybrid automata. In: HSCC. Volume 2289 of LNCS., Springer (2002) 465–478
9. Chutinan, A., Krogh, B.: Verification of infinite-state dynamic systems using approximate quotient transition systems. IEEE Transactions on Automatic Control **46** (2001) 1401–1410
10. Clarke, E., Gupta, A., Kukula, J., Strichman, O.: Sat based abstraction-refinement using ilp and machine learning techniques. In: CAV. LNCS, Springer (2002)
11. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: PLDI. SIGPLAN 36(5) (2001)
12. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Symp. on Principles of Programming Languages, ACM Press (2002) 58–70
13. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
14. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Stursberg, O., Theobald, M.: Verification of hybrid systems based on counterexample-guided abstraction refinement. In: Technical Report. (2002) Downloadable from *http://www.cs.cmu.edu/~theobald.*

15. Lafferriere, G., Pappas, G., Yovine, S.: A new class of decidable hybrid systems. In: HSCC. LNCS 1569, Springer (1999) 103–116
16. Henzinger, T., Kopke, P., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: Symposium on Theory of Computing, ACM Press (1995) 373–382
17. Dang, T., Maler, O.: Reachability analysis via face lifting. In: HSCC. LNCS 1386, Springer (1998) 96–109
18. Chutinan, A., Krogh, B.: Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In: HSCC. LNCS 1569, Springer Verlag (1999) 76–90
19. Greenstreet, M., Mitchell, I.: Reachability analysis using polygonal projections. In: HSCC. LNCS 1569, Springer (1999) 103–116
20. Kurzhanski, A., Varaiya, P.: Ellipsoidal techniques for reachability analysis. In: HSCC. LNCS 1790, Springer (2000) 203–213