SAT based Predicate Abstraction for Hardware Verification *

Edmund Clarke[1], Muralidhar Talupur[1], Helmut Veith[2], and Dong Wang[1]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, USA
{emc,tmurali,dongw}@cs.cmu.edu
[2] Institut für Informationsysteme, Technische Universität Wien, Vienna, Austria
veith@dbai.tuwien.ac.at

Abstract. Predicate abstraction is an important technique for extracting compact finite state models from large or infinite state systems. Predicate abstraction uses decision procedures to compute a model which is amenable to model checking, and has been used successfully for software verification. Little work however has been done on applying predicate abstraction to large scale finite state systems, most notably, hardware, where the decision procedures are SAT solvers. We consider predicate abstraction for hardware in the framework of *Counterexample-Guided Abstraction Refinement* where in the course of verification, the abstract model has to be repeatedly refined. The goal of the refinement is to eliminate *spurious behavior* in the abstract model which is not present in the original model, and gives rise to false negatives (spurious counterexamples).

In this paper, we present two efficient SAT-based algorithms to refine abstract hardware models which deal with spurious transitions and spurious counterexamples respectively. Both algorithms make use of the conflict graphs generated by SAT solvers. The first algorithm extracts constraints from the conflict graphs which are used to make the abstract model more accurate. Once an abstract transition is determined to be spurious, our algorithm does not need to make any additional calls to SAT solver. Our second algorithm generates a compact predicate which eliminates a spurious counterexample. This algorithm uses the conflict graphs to identify the important concrete variables that render the counterexample spurious, creates an additional predicate over these concrete variables, and adds it to the abstract model. Experiments over hardware designs with several thousands of registers demonstrate the effectiveness of our methods.

* 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), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, the General Motors Collaborative Research Lab at CMU, the Austrian Science Fund Project N Z29-N04, and the EU Research and Training Network GAMES. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, DOD, ARO, or the U.S. government.

1 Introduction

Counterexample-Guided Abstraction Refinement. Model checking [11, 10] is a widely used automatic formal verification technique. Despite the recent advancements in model checking technology, practical applications are still limited by the state explosion problem, i.e., by the combinatorial explosion of system states. For model checking large real world systems, symbolic methods such as BDDs and SAT-sovlers need to be complemented by abstraction methods [6, 5]. By a conservative abstraction we understand a (typically small) finite state system which preserves the behavior of the original system, i.e., the abstract system allows more behavior than the original (concrete) system. If the abstraction is conservative then we have a preservation theorem to the effect that the correctness of universal temporal properties (in particular properties specified in universal temporal logics such as $ACTL^*$) on the abstract model implies the correctness of the properties on the concrete model. In this paper, we are only concerned with universal safety properties whose violation can be demonstrated on a finite counterexample trace. The simplest and most important examples of such properties are system invariants, i.e., ACTL* specifications of the form $\mathbf{AG}p$.

A preservation theorem only ensures that universal properties which hold on the abstract model are indeed true for the concrete model. If, however, a property is violated on the abstract model, then the counterexample on the abstract model may possibly not correspond to any real counterexample path. False negatives of this kind are called *spurious* counterexamples, and are consequences of the information loss incurred by reducing a large system to a small abstract one. Since the spurious behavior is due to the approximate nature of the abstraction, we speak of *overapproximation*.

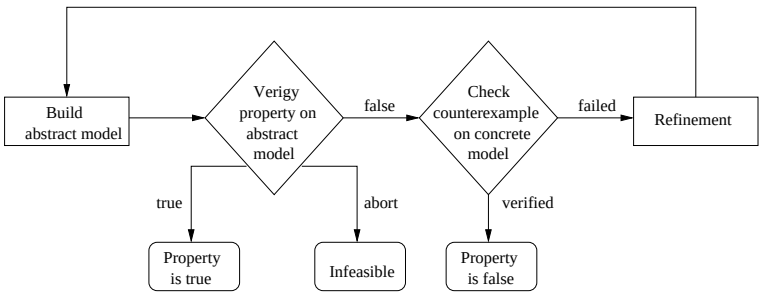

Fig. 1. Counterexample-Guided Abstraction Refinement.

In order to remedy the effect of abstraction and overapproximation, the abstraction needs to be refined in such a way that the spurious counterexample is eliminated. *Counterexample guided abstraction refinement* (CEGAR) [7] automates this procedure. CEGAR analyzes the spurious abstract counterexample to find a refinement of the current abstraction, such that the spurious counterexample is eliminated from the refined abstract model. This procedure is repeated until the property is either confirmed or

refuted, cf. Figure 1. During the last years, CEGAR Methods have found many applications in verification. See [9] for an overview.

Predicate Abstraction. Predicate abstraction [17, 1, 13, 12, 16, 18] is a specific construction to obtain conservative abstractions. In predicate abstraction, we identify a set of predicates $P_1, \ldots, P_m$ which describe important properties of the concrete system. The predicates are given by formulas which depend on the variables of the concrete system. The crucial idea is now to construct 2^m abstract states such that each abstract state corresponds to one valuation of the predicates, and thus to a conjunction of literals over $B_1, \ldots, B_m$.

More formally, we use the predicates P_i 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 is labeled with all the predicates it satisfies. The abstract state space contains one boolean variable B_j for each predicate P_j. An abstract state is labeled with predicate P_j if the corresponding Boolean variable B_j has value 1 in this state.

The predicates are also used to define a relation 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 $\hat{s}$, i.e.,

$$\rho(s, \hat{s}) \quad = \quad \bigwedge_{1 \leq j \leq m} P_j(s) \Leftrightarrow B_j(\hat{s}).$$

We now define the *concretization function* γ, which maps a set of abstract states to the corresponding set of concrete states. Each set of abstract states can be described by a Boolean formula over the variables $B_1, \ldots, B_m$. For a propositional formula $\hat{f}$ over the abstract state variables, we define $\gamma(\hat{f}) = \hat{f}[B_j \leftarrow P_j]$ to be the formula obtained by replacing each occurrence of some B_j by the corresponding P_j. The abstract initial states $\hat{S}_0$ and the abstract transition relation $\hat{R}$ are defined as

$$\hat{S}_0 \quad = \quad \bigwedge \{\hat{Y}_1 \mid S_0 \rightarrow \gamma(\hat{Y}_1)\} \tag{1}$$

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

where $\hat{Y}$ ($\hat{Y}_1$) denotes conjunctions (disjunctions resp.) of the literals over the current abstract state variables $\{B_1, \ldots, B_m\}$ and $\hat{Y}'$ denotes disjunctions 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*. In the most accurate 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.

Building the most accurate abstract model is expensive because the number of implications that need to be checked in the worst case is exponential in the number of predicates. Thus, to reduce the abstraction time, in practice an *approximate abstract model* is constructed by intentionally excluding certain implications from consideration. Therefore, there will be more behaviors in this approximate model than in the most accurate abstract model. We call those 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 preserves all behaviors of the original concrete system, the preservation theorem still holds.

Contribution. 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 either cannot handle infinite state systems, or only very specific ones. Predicate abstraction can extract finite state abstract models which are amenable to model checking from infinite state systems. Since hardware systems are finite state, model checking, possibly combined with simple forms of abstraction such as the localization reduction [14]) has been traditionally used to verify them. Existing predicate abstraction techniques for verifying software however 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 decision procedures or 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.

When refining the abstract model, we distinguish two cases of spurious behavior:

1. **Spurious Transitions** are abstract transitions which do not have any corresponding concrete transitions. By definition, spurious transitions cannot appear in the most accurate abstract model.
2. **Spurious Prefixes** are prefixes of abstract counterexample paths which do not have corresponding concrete paths. These are the typical spurious counterexamples described in the literature.

Our first SAT based algorithm deals with the first case, i.e., spurious transitions. As argued above, it is time consuming to build the most accurate abstract model when the number of predicates is large. 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), and similarly 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. This constraint however should be made as general as possible so that many related spurious transitions are removed simultaneously. An algorithm for this has been proposed in [12] which in the worst case requires $2m$ number of calls to a theorem prover, where m is the number of predicates. In this paper, 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 [12]. We give a detailed description in Section 3.1.

Even after removing spurious transitions the abstract counterexample can have a spurious prefix. This happens when the set of predicates is not rich enough to capture the relevant behaviors of the concrete system, even for the most accurate abstract model. In this case, a new predicate is identified and added to the current abstract model to invalidate the spurious abstract 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 proof obligations 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. The new predicate is then calculated by a projection-based SAT enumeration algorithm. Our experiments show that this algorithm can efficiently compute the required predicates for designs 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, there are several important differences: 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 [12]. 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 obtain the same constraints, probably using much less time, by combining the two algorithms together. Furthermore, the work in [12] does not consider the problem of introducing new predicates to refine the abstract model.

Other refinement algorithms in the literature compute new predicates using techniques such as syntactical transformations [16] or pre-image calculation [7, 18]. In contrast to this, 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.

More recently, [8] shows that predicate abstraction can benefit from localization reduction when verifying control intensive systems. By selectively incorporating state machines that control the behavior of the system under verification, more compact abstract models can be computed than those only based on predicate abstraction.

Outline of the paper. The rest of the paper is organized as follows. In Section 2 we describe the conflict dependency analysis. The refinement algorithms are presented in

Section 3. Section 4 contains the experimental results, and 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; we assume that the reader is familiar with the basic concepts of SAT solvers such as CHAFF [19]. As presented in [19], a conflict graph is an implication graph whose sink is the conflict vertex, and a conflict clause is obtained from a vertex cut of the conflict graph that separates the decision vertices from the conflict vertex.

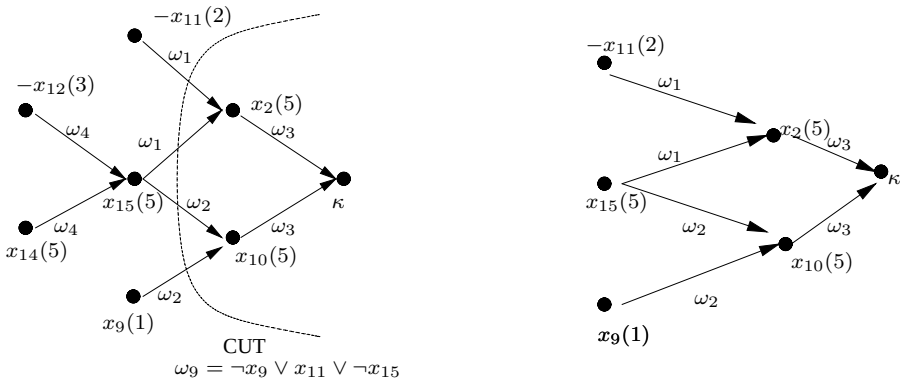

Fig. 2. Part of an implication graph and its corresponding G_{CUT}.

Let G be a conflict graph, κ be the conflict vertex in G, and CUT be a vertex cut which corresponds to the conflict clause $cl(CUT)$. Then G_{CUT} is the subgraph of G whose sources are the vertices in CUT and whose sink is κ. An example of a conflict graph and a vertex cut is given in Figure 2. For a subgraph G' of a conflict graph G, let $\Omega(G')$ be the set of clauses that occur as labels on the edges in the graph G'. Since G_{CUT} includes the conflict vertex κ, it is easy to see that $\neg cl(CUT) \wedge \Omega(G_{CUT}) \Rightarrow \mathrm{false}$. Therefore we obtain

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

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 $SUB(f) = dep(\theta)$ denote the clauses actually used for showing that f is unsatisfiable. Since $dep(\theta) \Rightarrow \theta$, $SUB(f) \subseteq f$ is also unsatisfiable, i.e.,

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

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 concrete run of the SAT solver during SAT search. Thus, for an unsatisfiable CNF formula f, $SUB(f)$ is in general not the minimal unsatisfiable subset of f; in many practical cases, however, $SUB(f)$ is significantly smaller than f.

3 Refinement for Predicate Abstraction

In this section, we describe the two SAT-based refinement algorithms for spurious transitions and spurious prefixes. To this end, we first need to introduce some more notation to represent the unrolling of a transition system from initial states.

Let V be the set of system variables, and V' be the corresponding set of primed variables used to describe the next state. Both V and V' are called *untimed variables*. For every variable v in V we maintain a version v^i of that variable for each time point $i \geq 0$. V^i, is the set of timed versions of variables in V at time $i \geq 0$. The variables V^i are called *timed variables* at time i.

Let $f(V)$ be a boolean function describing a property of concrete states; thus, it 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 defined over the timed variables V^i. We define an operator, called utf ("untimed function") which for a given timed function $f^i(V^i)$ returns the untimed function $f(V)$, i.e, $f(V) = utf(f^i(V^i))$.

Similarly, given a relation $r(V, V')$, which relates a current state over variables V to the next state over 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 $B = \{B_1, \ldots, B_m\}$ be the set of abstract state variables, and B^i be the corresponding timed variables. Given a timed abstract expression f ranging over variables 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 P^i_j, the timed version of the corresponding predicate.

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 each $ce_i(B^i)$ is a cube (i.e., a conjunction of literals) over the set of abstract variables B^i at time i. When it is clear from the context, we sometimes represent a counterexample without explicitly mentioning timed variables.

Let $ce = \langle ce_0, ce_1, \ldots, ce_n \rangle$ be an abstract counterexample, and i 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 correspond to the prefix of the abstract counterexample up to time i is given by

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

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, i.e., the conjunction obtained from c by removing all literals based on atoms in $BV \setminus BV_1$.

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 $SA[BV](f)$, cf. [15]. This SAT enumeration technique works as follows: Once a satisfiable solution is found, a blocking clause over BV_1 is created to avoid regenerating the same projected solution. After this blocking clause is added, the SAT search continues, pretending that no solution was found. 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 the 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.

Recall from the above discussion that an abstract counterexample $ce = \langle ce_0, ce_1, \ldots, ce_n \rangle$ corresponds to a real counterexample if and only if the set $prf(n)$ is not empty. If the abstract counterexample is a real counterexample, then by virtue of the preservation theorem, the property to be verified is also false for the concrete model. Otherwise, the counterexample is spurious and we need to refine the current abstract model. As argued above, there are two possible reasons for the existence of a spurious counterexample: One is that the computed abstract model is a too coarse overapproximation of the most accurate abstract model. The other possibility is that the set of predicates we used is not rich enough to model the relevant behaviors of the system, and thus we have a spurious prefix. In Section 3.1, we describe how our algorithm deals with the first case. In Section 3.2 we deal with the case where the set of predicates has to be extended.

3.1 Refinement for Spurious Transitions

Let $ce = \langle ce_0, ce_1, \ldots, ce_n \rangle$ be an abstract counterexample as defined above. If there exists an index 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*. This means that there are no concrete transitions which correspond to the abstract transition from ce_{i-1} to ce_i. It is evident that in this case the abstract counterexample does not have a corresponding real counterexample.

Recall that in the most accurate abstract model, there is at least one concrete transition corresponding to every abstract transition; consequently, the 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 $\hat{R}$.

To determine whether $trans(i-1, i)$ is empty or not, we convert it into a SAT unsatisfiability problem. For the spurious transition from ce_{i-1} to ce_i, we have

$$R^{i-1} \wedge \gamma(ce_{i-1}) \wedge \gamma(ce_i) \quad \Leftrightarrow \quad \text{false},$$

and 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 $\gamma(ce_i)$, a natural (but naive) approach is to add the constraint

$$utr(ce_{i-1} \rightarrow \neg ce_i)$$

to $\hat{R}$. It is evident that the resulting transition relation is correct and eliminates the spurious transition.

The constraint $ce_{i-1} \rightarrow \neg ce_i$ however can potentially involve a much larger number of the abstract state variables than necessary; it will therefore be very specific and not very useful in practice. It is thus reasonable to make the constraint as general as possible as long as 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 from the constraint $ce_{i-1} \rightarrow \neg ce_i$ in order to obtain a more general constraint.

Computing a General Constraint. Let m be the number of predicates. The problem of finding a general constraint which eliminates a spurious transition can be viewed as follows: Given propositional formulas f and f_j, $1 \leq j \leq 2m$, which make

$$f \wedge \bigwedge_{1 \leq j \leq 2m} f_j$$

unsatisfiable, we need to find a small subset $care \subseteq \{1, \ldots, 2m\}$, such that $f \wedge \bigwedge_{j \in care} f_j$ is unsatisfiable.

Returning to our problem of computing a general constraint, it is easy to see that if we set $f = R^{i-1}$ 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 $ce_{i-1} \rightarrow \neg ce_i$. Consequently, 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 will be a clause T_j containing only the literal t_j. So, instead of keeping track of the f_j's directly we keep track of the clauses 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. Since $SUB(F)$ denotes the clauses used in the refutation of F, it is easy to see that we can set $care = \{j \mid T_j \in SUB(F)\}$ to obtain the indices for the relevant f_j's as desired. Using only the f'_js given by $care$, we can now 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. Using the approach in [12], a potentially more general constraint than the one computed by our algorithm can be found. It works by testing for each f_j whether removing it keeps the resulting formula unsatisfiable. Their algorithm however requires $2m$ calls to a theorem prover, which is time consuming when the number m of predicates 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. In practice, however, its size is comparable to a minimal set. Note that it is easy to modify our algorithm in such a way as to make $care$ minimal: After the set $care$ is computed, we can try to eliminate the remaining literals one by one as in [12], 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 for Spurious Prefixes

Even after we have ensured that there are no spurious transitions in the counterexample ce, the counterexample itself can still be spurious. Such a situation is possible because even in the absence of spurious transitions we only have the most accurate abstract model; even in the most accurate abstract model it is not necessarily the case that there exists a concrete path which corresponds to the abstract counterexample, cf. [7]. In the current section we deal with this case.

Let n be the length of the given abstract counterexample. We are interested in finding a 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, let us consider the states in V^{k-1} in more detail, adopting the terminology of [7]:

- The set of states $SA[V^{k-1}](prf(k-1))$ is called the set of *deadend states*, denoted by $deadend$. 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. 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. By construction we know that

$$deadend \neq \emptyset, \qquad bad \neq \emptyset, \qquad (deadend \cap bad) = \emptyset.$$

As pointed out in [7], it is impossible to distinguish between the states in $deadend$ and bad using the existing set of predicates, because all involved states correspond to the same abstract state ce_{k-1}. Therefore, our refinement algorithm aims to find a *new* separating predicate sep, such that

$$deadend \subseteq sep \qquad \text{and} \qquad sep \cap bad = \emptyset.$$

After introducing sep as a new predicate, the abstract model will be able to distinguish between the deadend and bad states. Note that we can also use an alternative symmetric definition for sep which satisfies $bad \subseteq sep$ and $deadend \cap sep = \emptyset$.

We call the set of concrete state variables over which a predicate is defined the *support* of the predicate. In order to compute a predicate sep with minimal support, we will describe an algorithm which 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. The details of this procedure are described in the rest of this section.

Minimizing the Support of the Separating Predicate. An important goal of our refinement algorithm is to compute a *compact predicate*, i.e., a predicate that can be represented compactly. For large scale hardware designs, existing refinement algorithms such as weakest precondition calculation, preimage computation, syntactical transformations etc., may fail because the predicates they aim to compute become too big. Our algorithm avoids this problem by first identifying a minimal set of concrete state variables that are responsible for the failure of the spurious prefix. Our algorithm guarantees that there exists a separating predicate over this minimal set that can separate the deadend and bad states. It is reasonable to assume that a predicate with a small support has a compact representation.

Our algorithm for computing the support of sep is similar to the one used in finding the important registers for the localization reduction in [2]. By assumption, the CNF formula for $prf(k)$ is unsatisfiable. Thus, we can use the conflict dependency analysis from Section 2 to identify an unsatisfiable subset $SUB(prf(k))$ of the clauses in $prf(k)$. Let $\mu(ce, k-1)$ denote the concrete state variables at time $k-1$ whose CNF variables are in $SUB(prf(k))$, i.e.,

$$\mu(ce, k-1) = ssuppt[V^{k-1}](SUB(prf(k))).$$

When the context is clear we will for simplicity refer to $\mu(ce, k-1)$ as μ. Let $deadend_\mu = \texttt{proj}[\mu](deadend)$ be the projection of the deadend states on μ, and $bad_\mu = \texttt{proj}[\mu](bad)$ be the projection of the deadend states on μ. By the definition of SUB and μ it follows that

$$\mu \neq \emptyset \qquad \text{and} \qquad deadend_\mu \cap bad_\mu = \emptyset. \tag{8}$$

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. As argued above, any set of concrete states that separates $deadend_\mu$ and bad_μ is a 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 proceeds in several steps.

- First, we try to compute bad_μ using a SAT enumeration algorithm, which avoids computing bad first. To this end, the set of bad states bad is converted to a CNF formula. Once a satisfying assignment is found, we project the satisfying assignment to μ and add it into the set of solutions. A blocking clause over μ is also added to the set of clauses and the SAT search is continued. This procedure repeats until there are no more solutions. The collected set of solutions over μ is naturally represented in DNF. 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. Note that $\neg bad_\mu$ 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. Note that it is possible that the set ODE overlaps with bad_μ. We define $SODE = \texttt{proj}[\mu](ODE \wedge bad)$ as the intersection of the two sets. Then the desired separating predicate is given by

  $$ODE \wedge \neg SODE,$$

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.
The over-approximation of $deadend_\mu$ is computed by a projection-based method: We partition the variables in μ into smaller sets $\mu_1, \ldots, \mu_l$ based on the closeness of the variables, whereby the criterion for closeness is based on circuit structure [3]. Since each set μ_i is small, we can compute each $deadend_{\mu_i}$ easily. The over-approximation is then given by

$$ODE = \bigwedge_{1 \le i \le l} deadend_{\mu_i}.$$

– If in a rare case, even $SODE$ can not be efficiently enumerated using SAT we identify important registers using the algorithms in [2], and add them as a new predicate to make sure the abstract model is refined. We did not encounter this case in any of our experiments.

After the calculated separating predicate sep is added as a new predicate, we introduce a new abstract Boolean variable B_{m+1}. for sep. Then we add the constraint $B_{m+1} \rightarrow utr(ce_{k-1} \rightarrow \neg ce_k)$ to the abstract transition relation. Thus, the spurious counterexample is eliminated in the refined abstract model.

4 Experimental Results

We have implemented our predicate abstraction refinement framework on top of the NuSMV model checker [4]. We modified the SAT checker zChaff [19] to support conflict dependency analysis. We also developed a Verilog parser to extract useful predicates from Verilog designs directly; due to lack of space, we omit a detailed description of this parser. 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 inte-

circuit	# regs	# gates	ctrex	Localization			Predicate Abstraction		
			length	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.

ger 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** p properties where p specifies a certain combination of values for several control registers. For all the properties shown in the first column of Table 1, we have performed the 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 [7], 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 several hundred 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 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.

To create the abstract transition relation in our experiments, we only considered implications in equation (2) where the left hand side has at most 2 literals, and the right hand side has 1 literal, and relied on the refinement algorithm to make the abstract transition relation as accurate as necessary. In all our experiments, our projection based SAT enumeration algorithm can successfully create the new predicates when necessary. Thus we never have to resort to other methods for creating new separating predicates.

5 Conclusion

We have presented two SAT-based counterexample guided refinement algorithms to enable efficient predicate abstraction of large hardware designs. In order to reduce the abstraction time, we initially construct an approximate abstract model which may lead not only to spurious counterexamples, but also to spurious transitions, i.e., abstract transitions which do not have a corresponding concrete transition. Our first SAT based refinement algorithm is used to eliminate spurious transitions. Our second SAT based refinement algorithm eliminates spurious counterexample prefixes. It extends the predicate abstraction by a new predicate ranging over a minimal number of state variables. The predicates computed by our algorithm can be represented compactly as CNF formulas or conjunctions of DNF formulas. Our experimental results demonstrate 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. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics, 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 176–194, 2001.
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*, 2000. Extended Version to appear in J.ACM.
8. Edmund Clarke, Orna Grumberg, Muralidhar Talupur, and Dong Wang. High level verification of control intensive systems using predicate abstraction. In *MEMOCODE*, 2003.
9. Edmund Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In *Proc. International Symposium on Verification in Honor of Zohar Manna*, volume 2772 of *LNCS*, 2003.
10. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
11. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs, volume 131 of Lect. Notes in Comp. Sci.*, pages 52–71, 1981.
12. S. Das and D. Dill. Successive approximation of abstract transition relations. In *LICS'01*, 2001.
13. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *CAV'99*, pages 160–171, 1999.
14. R. P. Kurshan. *Computer-Aided Verification*. Princeton Univ. Press, Princeton, New Jersey, 1994.
15. K. McMillan. Applying SAT methods in unbounded symbolic model checking. In *CAV'02*, pages 250–264, 2002.
16. K. Namjoshi and R. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV'00*, 2000.
17. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV'97*, pages 72–83, 1997.
18. H. Saidi and N. Shankar. Abstract and model check while you prove. In *CAV'99*, pages 443–454, 1999.
19. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *ICCAD'01*, 2001.