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, 9, 10, 12, 15, 17, 18], 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 [5, 8, 14], which is a conservative approach for model checking universal temporal logic [8] 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 [6]. To get rid of a spurious counterexample, the abstraction needs to be made more precise via refinement. *Counterexample guided abstraction refinement* [6, 12, 19] (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 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.

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 [8] 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* [8])*
Let $M = (S, S_0, R, L)$ and $\hat{M} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$ be two transition systems, with A and $\hat{A}$ as the respective sets of atomic propositions and let $\rho \subseteq S \times \hat{S}$ be a relation such that $M \preceq_\rho \hat{M}$. Then, for any ACTL formula, Φ with atomic propositions in $A \cap \hat{A}$*

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

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* [5, 14].

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,

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

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,

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

In predicate abstraction [18], the abstract initial states $\hat{S}_0$ and the abstract transition relation $\hat{R}$ are defined as

$$\hat{S}_0 = \bigwedge \{\hat{Y}_1 \mid S_0 \rightarrow \gamma(\hat{Y}_1)\} \tag{6}$$

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

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 [16], 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, 18]. 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 [13] 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

$$\hat{S}_0 = \wedge_{v \in U} I_v \tag{8}$$

$$\hat{R} = \wedge_{v \in U} R_v \tag{9}$$

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 [18]. 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 [11]. 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, 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.

```
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

It is usually the case that there are many branch conditions that we can extract 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

	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

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 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.

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.

```
    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

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 [6] using SAT. Details can be found in [7].

```
    //$i$: 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}

If the counterexample is invalidated at line (1) in Figure 4, the algorithm in Figure 5 is called with the time t

```
    //$t$: 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

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 [15], 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 [6]. They extract conditions of **case** statements in the SMV language in order to build the initial abstraction. The extraction method in [6] 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 [12]. The goals of their algorithm and ours are different. In [12], 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 modified the zChaff SAT solver [20] to generate proofs of unsatisfiability. The techniques in this paper are implemented on top of the predicate abstraction framework reported in [7]. We compare results with and without these techniques using 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 [7] 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).

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

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 [7] and with our techniques

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 *FM-CAD'02*, 2002.

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

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

[7] Edmund Clarke, Muralidhar Talupur, and Dong Wang. SAT based Predicate Abstraction for Hardware Verification. In *Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.

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

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

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

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

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

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

[14] 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.

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

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

[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] 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.

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