Making Predicate Abstraction Efficient:
How to eliminate redundant predicates [⋆]

Edmund Clarke[1], Orna Grumberg[2]
Muralidhar Talupur[1], Dong Wang[1]

[1] Carnegie Mellon University [2] TECHNION - Israel Institute of Technology

Abstract. In this paper we consider techniques to identify and remove redundant predicates during predicate abstraction. We give three criteria for identifying redundancy. A predicate is redundant if any of the following three holds (i) the predicate is equivalent to a propositional function of other predicates. (ii) removing the predicate preserves safety properties satisfied by the abstract model (iii) removing it preserves bisimulation equivalence. We also show how to efficiently remove the redundant predicates once they are identified. Experimental results are included to demonstrate the effectiveness of our methods.

Keywords: predicate abstraction, redundancy, simulation, bisimulation, safety properties

1 Introduction

Abstraction has been widely accepted as a viable way for reducing the complexity of systems during temporal logic model checking [10]. Predicate abstraction [1–3, 11, 12, 14, 19, 21, 22] has emerged as one of the most successful abstraction techniques. It has been used in both software and hardware verification. In this paper, we give a technique to improve predicate abstraction by eliminating redundant predicates. This technique can be applied in both software and hardware verification. We give efficient verification algorithms for finite state systems (hardware) and outline how our method can be used for infinite state systems (software).

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

Counterexample guided abstraction refinement (CEGAR) [7, 16, 20] is an example of a commonly used abstraction technique. It works by introducing new predicates to eliminate spurious counterexamples. The new predicates depend on certain abstract states in the spurious abstract counterexample. Thus, different predicates are likely to be closely related when similar

[⋆] 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.

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

Removing a predicate involves constructing the abstract model using the reduced predicate set. We give a simple method to construct the reduced abstract model from the original abstract model in Section 4.

1.1 Related Work

The notion of redundancy has been explored in resolution theorem proving [5], where it is called *subsumption*. Intuitively a clause is considered redundant if it is logically implied via substitution by other clauses. Our conditions for redundancy are more complicated. Even if a predicate is implied by other predicates, we still need to consider the abstract transition relation in order to decide whether removing the predicate will affect the results of verifying a given property.

The work that is closest to ours is the notion of *strengthening* in [1]. To build the abstract model, the weakest precondition is converted to an expression over the set of predicates in the abstraction. Thus strengthening is somewhat similar to the *replacement function* in this paper. However, in [1], the result of the strengthening is over all the predicates, while the replacement function used here is defined over a subset of the predicates. Finally, the two transformations have different purposes. Strengthening is only used to build an abstract model; while our transformation is used to remove redundant predicates and thus reduce the complexity of the abstract model.

This paper removes unnecessary predicates introduced by counterexample guided refinement. Recently, abstraction refinement with more than one counterexamples has been investigated in [18, 13]. However, there is no guarantee for the elimination of redundant predicates by considering multiple counterexamples in computing predicates alone. Thus, our techniques can also be applied in that context.

Exploiting functional dependencies between state variables to reduce BDD size has been investigated in [15]. In that approach, if a variable can be shown to be a function of other variables, it can be eliminated during BDD based verification. Our approach is more general in that it is possible to remove a predicate even if it is not equivalent to any function over other predicates.

1.2 Outline of the Paper

In the next section we introduce predicate abstraction and other relevant theory. In Section 3 we define the *replacement function* and show how to compute it. In the next section we show how to construct the new abstract model after removing a redundant predicate. In Section 5, the simplest form of redundancy, called equivalence induced redundancy, is presented. Section 6 and Section 7 give redundancy conditions that preserve safety properties and bisimulation equivalence respectively. The comparison between redundancies in the last two sections is illustrated using examples in Section 8. We discuss how our algorithms can be applied to software verification in Section 9. In Section 10, we describe our experiments. Section 11 concludes the paper with some directions for future research.

2 Preliminaries

In this section we review the relevant theory of property preserving abstractions introduced by Clarke, Grumberg and Long in [6] and Loiseaux et al. in [17]. Using this theory we describe the predicate abstraction framework of Saidi and Shankar [22].

2.1 Notation

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

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

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

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

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

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

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

If relation ρ is a total function on S_1, then $\widetilde{pre}[\rho]$ is the same as $pre[\rho]$ [17].

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

Definition 1. *Let Id_S denotes the identity function on the powerset of* S. *A* Galois connection *between 2^{S_1} and 2^{S_2} is a pair of monotonic functions (α, γ), where $\alpha : 2^{S_1} \rightarrow 2^{S_2}$ and $\gamma : 2^{S_2} \rightarrow 2^{S_1}$, such that $Id_{S_1} \subseteq \gamma \circ \alpha$ and $\alpha \circ \gamma \subseteq Id_{S_2}$.*

The functions α and γ are often called the *abstraction function* and the *concretization function*, respectively. The Galois connection that we will be using in this paper is described in the following proposition.

Proposition 1. *[17] Given a relation $\rho \subseteq S_1 \times S_2$, the pair $(post[\rho], \widetilde{pre}[\rho])$ is a Galois connection between 2^{S_1} and 2^{S_2}.*

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

2.2 Existential Abstraction

We model circuits and programs as transition systems. Given a set of atomic propositions, *A*, let $M = (S, S_0, R, L)$ be a *transition system* (refer to [9] for details).

Definition 2. *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 1. *(Preservation of ACTL* [9])*
Let $M = (S, S_0, R, L)$ and $\hat{M} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$ be two transition systems, with A and $\hat{A}$ as the respective sets of atomic propositions and let $\rho \subseteq S \times \hat{S}$ be a relation such that $M \preceq_\rho \hat{M}$. Then, for any ACTL formula, Φ with atomic propositions in $A \cap \hat{A}$*

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

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 in $pre[\rho](\hat{s})$ have the same labeling over a subset $\hat{A}$ of A. Then an abstract transition system $\hat{M} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$ over $\hat{A}$ which simulates M can be constructed as follows:

$$\hat{S_0} = post[\rho](S_0) = \exists s.\ S_0(s) \wedge \rho(s, \hat{s}) \tag{1}$$
$$\hat{R}(\hat{s}, \hat{s}') = \exists s\ s'.\ \rho(s, \hat{s}) \wedge \rho(s', \hat{s}') \wedge R(s, s') \tag{2}$$
$$\text{for each } \hat{s} \in \hat{S}, \hat{L}(\hat{s}) = \bigcap_{s \in pre[\rho](\hat{s})} (L(s) \cap \hat{A}) \tag{3}$$

Proposition 2. *For M and $\hat{M}$ in the above construction $M \preceq_\rho \hat{M}$*

In the above construction $\hat{R}$ is defined in terms of the abstract current state $\hat{s}$ and the abstract next state $\hat{s}'$. This construction is from [6], and it is also implicit in the paper by Loiseaux et al. [17]. This kind of abstraction is called *existential abstraction*. The set of initial states in the abstract system are those states of $\hat{S}$ that are related to the initial states of M. Note that for any two states *s* and $\hat{s}$ related under *ρ* the property $L(s) \cap \hat{A} = \hat{L}(\hat{s})$ holds.

2.3 Predicate Abstraction

Predicate abstraction 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,

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

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. Based on Section 2.1, the pair of functions $post[\rho]$ and $\widetilde{pre}[\rho]$ generated from relation ρ forms a Galois connection. We will denote this Galois connection by (α, γ). Note that since ρ is a total function, $\widetilde{pre}[\rho] = pre[\rho]$. Using this ρ and the construction given in the previous subsection, we can build an abstract model which simulates the concrete model. In [22] the abstract transition relation $\hat{R}$ is defined as

$$\bigwedge\{\hat{Y} \to \hat{Y}' \mid \forall V,\ V'.((R(V, V') \wedge \gamma(\hat{Y})) \to \gamma(\hat{Y}'))\} \tag{5}$$

where $\hat{Y}$ is an arbitrary conjunction of the literals of the current state variables $\{B_1, B_2, \ldots, B_k\}$ and $\hat{Y}'$ is an arbitrary disjunction of literals of the next state variables $\{B'_1, B'_2, \ldots, B'_k\}$. It can be shown that (5) is equivalent to (2).

3 The Replacement Function

Our goal is to eliminate B_i from the abstract model $\hat{M}$ without sacrificing accuracy. For this purpose, we define an under-approximation, $\mathcal{F}_U(B_i)$, for B_i in terms of the other variables. More precisely, let M be a concrete transition system, $\{P_1, P_2, .., P_k\}$ be a set of predicates defined on the states of M, and let ρ be a total function defined by equation (4). Also, let $\hat{M}$ be the corresponding abstract transition system over $V = \{B_1, B_2, .., B_k\}$. The support of an abstract set of states $\hat{S}_1$ includes B_i if and only if

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

Consider the boolean variable B_i and the set $U = V \setminus \{B_i\}$. Let Φ denote either B_i or $\neg B_i$. The *replacement function* for Φ, denoted by $\mathcal{F}_U(\Phi)$, is defined as the largest set of *consistent* abstract states (we call an abstract state *consistent* if its concretization is not empty) whose support is included in U and whose concretization is a subset of $\gamma(\Phi)$. The implications $\gamma(\mathcal{F}_U(B_i)) \to \gamma(B_i)$ and $\gamma(\mathcal{F}_U(\neg B_i)) \to \gamma(\neg B_i)$ follow from this definition. Figure 1 shows the relationship between the concretization of a predicate B_i, $\mathcal{F}_U(B_i)$, and $\neg\mathcal{F}_U(\neg B_i)$.

We now show how to compute $\mathcal{F}_U(B_i)$. Consider the abstract state space $\hat{S}$ given by tuples of the form $(B_1, B_2, .., B_k)$. Not all the abstract states have corresponding concrete states. We

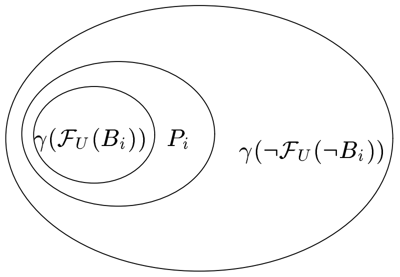
$\gamma(\mathcal{F}_U(B_i))$
P_i
$\gamma(\neg\mathcal{F}_U(\neg B_i))$

Fig. 1. Relationship between the concretization of B_i, $\mathcal{F}_U(B_i)$, and $\neg\mathcal{F}_U(\neg B_i)$

consider only the set of consistent abstract states, g, that are related to some concrete states by the relation (4) in Section 2.3. Formally, if S is the set of concrete states, $\{P_j | 1 \leq j \leq k\}$ is the set of predicates and ρ is the simulation relation as in Section 2.3, then

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

For hardware verification, all the concrete state variables have finite domain. The set g can be efficiently computed using OBDDs [4] through a series of conjunction and quantification operations. However, a general theorem prover is needed for infinite state systems. We define $g|_{B_i}$ to be the set of reduced abstract states obtained by taking all the states in g that have the bit B_i equal to 1 and dropping the bit B_i. Similarly $g|_{\neg B_i}$ is obtained by taking all those states in g with bit B_i equal to 0 and dropping the bit B_i. The following theorem shows that the set $(g|_{B_i} \wedge \neg g|_{\neg B_i})$ is a candidate for $\mathcal{F}_U(B_i)$.

Theorem 2. *Let $V = \{B_1, B_2, \ldots, B_k\}$ be the boolean variables. Let $U = V \setminus \{B_i\}$, and let $f_1 = g|_{B_i} \wedge \neg g|_{\neg B_i}$ be a set of abstract states. Then $\gamma(f_1) \Rightarrow \gamma(B_i)$ and f_1 is the largest set of consistent abstract states that does not have bit B_i in its support. Likewise, if $f_2 = g|_{\neg B_i} \wedge \neg g|_{B_i}$, then $\gamma(f_2) \Rightarrow \gamma(\neg B_i)$ and f_2 is the largest set of consistent abstract states that does not have bit B_i in its support.*

Replacement function is used extensively in the later sections. The correctness of our algorithms only depend on the property that $\gamma(\mathcal{F}_U(B_i)) \rightarrow \gamma(B_i)$. The nice advantage of this is we can use any f that satisfies $\gamma(f) \rightarrow \gamma(B_i)$ instead of using $\mathcal{F}_U(B_i) = f_1$, which is difficult to compute when there are many predicates. Instead we use the following approximation: we first partition predicates into clusters as in Section 5, then compute the set of consistent abstract states and replacement function for each cluster separately. We use these easy to compute approximations to identify and remove redundant predicates. This does not affect the correctness (i.e., every identified predicate is indeed redundant), but some redundant predicates may fail to be identified.

4 Removing Redundant Predicates

Removal of a predicate involves constructing a new abstract transition system from the old abstract transition system. The state space of the new abstract transition system is the set of all possible valuations of the boolean variables corresponding to the new predicate set. The new predicate set has one less predicate than the old predicate set. Let P_i be the redundant predicate that is to be removed. If the old state space is given by k-tuples $(B_1, B_2, ..., B_k)$, then the new state space is given by *(k-1)*-tuples $(B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_k)$. Suppose the original abstract model is $\hat{M} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$. We now describe how to construct the new abstract model, $M_r = (S_r, S_{0r}, R_r, L_r)$ (r for "reduced"), from $\hat{M}$ if we decide to drop the predicate P_i. The relation ρ_r between the concrete state space and the reduced state space is

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

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

$$R_r(s_r,\ s'_r) = \exists b_i, b'_i.\ \hat{R}(\langle s_r, b_i \rangle, \langle s'_r, b'_i \rangle) \tag{6}$$

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

$$S_{0r}(s_r) = \exists b_i.\ \hat{S}_0(\langle s_r, b_i \rangle) \tag{7}$$

Lemma 1. *The transition relation of the reduced abstract model defined by equation (6) is the same as the one built directly from the concrete model using equation (2) and ρ_r over the reduced set of predicates.*

Thus, R_r constructed using equation (6) is equivalent to the one constructed directly from the concrete model using equation (2).

5 Equivalence Induced Redundancy

In this section, we present the simplest form of redundancy, called *equivalence induced redundancy*. More specifically, a predicate P_i is redundant if there exist two propositional formulas in terms of other predicates that are logically equivalent to P_i and $\neg P_i$ respectively. The reduced abstract model can be built by replacing B_i and $\neg B_i$ using the equivalent formulas. It is easy to see that the resulting reduced abstract model is bisimilar to the original model. We present a method, based on replacement function, to determine if a predicate P_i can be expressed in terms of the other predicates. The following theorem shows that under some conditions the concretization of the replacement functions for B_i and $\neg B_i$ are logically equivalent to P_i and $\neg P_i$.

Theorem 3. *For a predicate P_i, if $g|_{B_i} \wedge g|_{\neg B_i} = \emptyset$, then $\gamma(\mathcal{F}_U(B_i)) \equiv P_i$ and $\gamma(\mathcal{F}_U(\neg B_i) \equiv \neg P_i$.*

Equivalence induced redundancy occurs often because of a heuristic we use in predicate abstraction. For hardware verification, the predicates we consider are all propositional formulas over concrete state variables with finite domains. It is well known that the number of propositional formulas over n boolean variables is 2^{2^n}. Therefore, it is possible that the abstract model is much bigger than the concrete model. So we use a heuristic to avoid this problem. Given a predicate, which is a propositional formula, the concrete state variables that this formula depends on are called the *supporting variables* of the predicate. We partition the predicates into clusters. Two predicates go into the same cluster if they share many supporting variables. Let c be a cluster. If the number of predicates in c is greater than the number of supporting variables (non-boolean variables are expanded to bits that are required to encode their domains) in c then we will use the supporting variables instead of the predicates to build the abstract model. So, all the original predicates in the cluster c become redundant. In this way, the overall size of the abstract model will be bounded by the size of the concrete model. Since software system may have unbounded state variables, this technique can not be used in general for software verification.

6 Redundant Predicates for Safety Properties

A predicate in a given set of predicates is *redundant* for a set of properties in $\mathcal{L}$ if the abstract transition system constructed without using this predicate satisfies the same set of properties as the original abstract transition system (constructed using all the predicates). In this section we deal with safety properties of the form $\mathbf{AG}\, p$, where p is a boolean formula without temporal operators. Note that any safety property can be rewritten into the above form through tableau construction with no fairness constraints [9].

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

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

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

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

where $\hat{s}[B_i \leftarrow 0]$ is a state that agrees with $\hat{s}$ on all bits except possibly the bit B_i, which is fixed to 0. $\hat{s}[B_i \leftarrow 1]$ is similar. Intuitively, if neither $\mathcal{F}_U(B_i)(\hat{s})$ nor $\mathcal{F}_U(\neg B_i)(\hat{s}))$ holds, the values of variables $B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_k$ can not determine the value of B_i. In order for $\hat{S}$ to be oblivious, it must contain states with both possible values of B_i.

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

$$\begin{aligned} \forall \hat{s}, \hat{s}'[\ \ & \neg\mathcal{F}_U(B_i)(\hat{s}') \wedge \neg\mathcal{F}_U(\neg B_i)(\hat{s}') \Rightarrow \\ & (\hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 1]) \Leftrightarrow \hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 0]))] \end{aligned} \tag{8}$$

In order to test whether a transition relation $\hat{R}$ is oblivious to B_i or not, we take the negation of (8) and formulate it as a SAT instance by converting it into a CNF formula. If the CNF formula is satisfiable then we conclude that $\hat{R}$ is not oblivious otherwise it is. The negation of (8) is the following

$$\begin{aligned} \exists \hat{s}, \hat{s}'[\ \ & \neg\mathcal{F}_U(B_i)(\hat{s}') \wedge \neg\mathcal{F}_U(\neg B_i)(\hat{s}') \wedge \\ & (\hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 1])\ \Leftrightarrow\ (\neg\hat{R})(\hat{s}, \hat{s}'[B_i \leftarrow 0]))] \end{aligned} \tag{9}$$

Theorem 4. *Given an abstract transition system $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$ which corresponds to a set of predicates V, and a safety property $f = \mathbf{AG}\,p$, where p is a propositional formula without temporal operators. Also assume that predicate B_i is one of the predicates in V but not one of the predicates in f. If $\hat{S}_0$ and $\hat{R}$ are oblivious to B_i, then the abstract transition system corresponding to the reduced set of predicates $U = V \setminus \{B_i\}$ satisfies f if and only if $\hat{M}$ satisfies it.*

7 Redundant Predicates for Bisimulation Equivalence

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

Let $\beta \subseteq \hat{S} \times S_r$ be a relation defined such that two states $\hat{s} \in \hat{S}$ and $s_r \in S_r$ are related under β if and only if $\hat{s} \in \mathit{extend}[B_i](s_r)$, where $\mathit{extend}[B_i](s_r)$ is as defined previously. We intend to make β a bisimulation relation between $\hat{M}$ and M_r. From the construction of M_r, it is easy to see that $\hat{M} \preceq_\beta M_r$. In order for $\hat{M}$ to simulate M_r, we must make sure that for any $b_i \in \{0, 1\}$, if $\langle s_r, b_i \rangle$ is a consistent abstract state, then $\langle s_r, b_i \rangle$ can simulate s_r. If only one of $\langle s_r, 0 \rangle$ and $\langle s_r, 1 \rangle$ is a consistent state, from (6), it is easy to see that any successor state of s_r corresponds to a successor of the single consistent state. In order to handle the case when both $\langle s_r, 0 \rangle$ and $\langle s_r, 1 \rangle$ are consistent, we have the following condition on $\hat{R}$: for any state $\hat{s} \in \hat{S}$

$$\neg \mathcal{F}_U(B_i)(\hat{s}) \wedge \neg \mathcal{F}_U(\neg B_i)(\hat{s}) \Rightarrow \forall \hat{s}'.(\hat{R}(\hat{s}[B_i \leftarrow 0], \hat{s}') \Leftrightarrow \hat{R}(\hat{s}[B_i \leftarrow 1], \hat{s}')) \qquad (10)$$

This condition says that if the value of B_i cannot be determined by the values of the other boolean variables, i.e., both $\hat{s}[B_i \leftarrow 0]$ and $\hat{s}[B_i \leftarrow 1]$ are consistent, then $\hat{R}$ does not distinguish between different values of the bit B_i. If $\mathcal{F}_U(B_i)(\hat{s})$ is true then we know that $\hat{s}[B_i \leftarrow 0]$ is inconsistent. If $\mathcal{F}_U(\neg B_i)(\hat{s})$ is true then we know that $\hat{s}[B_i \leftarrow 1]$ is inconsistent. In case that both of these are false (which is the condition on the left hand side of (10)), then we require that the successors of the states $\hat{s}[B_i \leftarrow 0],\ \hat{s}[B_i \leftarrow 1]$ be the same. Similar to Section 6, to test whether $\hat{R}$ satisfies condition (10) or not, we test the satisfiability of its negation.

$$\begin{aligned} \exists \hat{s}, \hat{s}'.\ \ & \neg \mathcal{F}_U(B_i)(\hat{s}) \wedge \neg \mathcal{F}_U(\neg B_i)(\hat{s}) \wedge \\ & (\hat{R}(\hat{s}[B_i \leftarrow 0], \hat{s}') \Leftrightarrow (\neg \hat{R})(\hat{s}[B_i \leftarrow 1], \hat{s}')) \end{aligned} \qquad (11)$$

Theorem 5. *If condition (10) holds, then β is a bisimulation relation between $\hat{M}$ and M_r*

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

8 Difference in the Bisimulation and $\mathbf{AG}\,p$ conditions

We have seen two redundancy conditions, one for preserving $\mathbf{AG}\,p$ properties and the other for preserving CTL^* properties. In this section, we give examples of transition relation which satisfy one of the conditions and violates the other. This demonstrates that the conditions (8) and (10) are not comparable.

8.1 A transition relation that satisfies the Bisimulation condition

We first present an abstract transition relation that satisfies the Bisimulation condition, (10), but does not satisfy the obliviousness condition required for preserving $\mathbf{AG}\,p$ properties. The abstract transition system is:

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

Suppose we are trying to remove B_2. Assume that $\mathcal{F}_U(B_2) = \neg B_3$ and $\mathcal{F}_U(\neg B_2) = \neg B_4$. The condition for bisimulation, (9), then is

$$\begin{array}{l}B_3 \wedge B_4 \Rightarrow [((B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow \\ \qquad ((B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4))]\end{array}$$

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

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

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

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

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

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

which is equivalent to

$$\begin{array}{l}B'_3 \wedge B'_4 \Rightarrow [((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow false)\ \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow \\ \qquad ((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1)\ \wedge (B_4 \rightarrow B'_4))]\end{array}$$

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

8.2 A transition relation that satisfies the AG p condition

The transition relation is

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

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

$$B'_3 \wedge B'_4 \Rightarrow [((B_3 \rightarrow \neg B'_4) \wedge (B_2 \rightarrow B'_1 \wedge B'_5) \wedge (\neg B_3)) \Leftrightarrow \\ ((B_3 \rightarrow \neg B'_4) \wedge (B_2 \rightarrow B'_1 \wedge B'_5) \wedge (B_3 \rightarrow B'_1))]$$

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

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

Hence the $\mathbf{AG}\, p$ preservation condition is satisfied. The bisimulation condition for this example (after some simplification) is:

$$B_3 \wedge B_4 \Rightarrow [((B_3 \rightarrow \neg B'_4) \wedge (B_3 \rightarrow B'_1 \wedge \neg B'_2)) \Leftrightarrow \\ ((B_3 \rightarrow \neg B'_4) \wedge (B'_1 \wedge B'_5) \wedge (B_3 \rightarrow B'_1 \wedge \neg B'_2))]$$

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

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

9 Removing Redundancy for Software Verification

In this section, we will show how our redundancy removal algorithms can be applied to software (or infinite state systems) verification. Performance of our redundancy removal algorithms depend on efficient computation of the replacement functions $\mathcal{F}_U(B_i)$ ($\mathcal{F}_U(\neg B_i)$) is important. Recall that, we first calculate the set of consistent abstract states g, and then use $g|_{B_i}$ and $g|_{\neg B_i}$ to define the replacement function. For software verification, predicates may be formulas involving unbounded state variables. Instead of using BDDs to compute g as in Section 3, we use a theorem prover to compute $g = \alpha(\textbf{true})$ as in traditional predicate abstraction [1, 21, 22]. Computing $\alpha(\textbf{true})$ could involve many calls to a theorem prover [22]. However, the correctness of our overall algorithm does not depend on the precise calculation of the set of consistent abstract states. Any approximation, f, of $\mathcal{F}_U(B_i)$ satisfying $\gamma(f) \Rightarrow \gamma(B_i)$ would be sufficient. So the existing techniques for approximating α can be applied [1].

Except for the replacement function, all the other computations, required in our algorithms to identify and remove redundancy, are performed on the original abstract model. It is usually the case that the abstract model is finite state. Therefore, the algorithms in the previous sections can be easily applied. The *boolean programs* in the SLAM project [1, 2], which have infinite control, are an exception. Extending our algorithms to them is left for future work.

10 Experimental Results

We have implemented our abstraction refinement framework on top of NuSMV model checker and the zChaff SAT solver [23]. The method to compute predicates for all the experiments is based on the deadend and bad states separation algorithm presented in [8].

We present the results for hardware verification in Table 1. We have 6 safety properties to verify for a programmable FIR filter (PFIR) which is a component of a system-on-chip design. For all the properties shown in the first column of Table 1, we have performed cone-of-influence reduction before the verification. The resulting number of registers and gates is shown in the second and third columns. Most properties are true, except for scr1 and prop5. The lengths of the counterexamples are shown in the fourth column. All these properties are difficult to verify for the state-of-art BDD-based model checker, Cadence SMV. Except for the two false properties, Cadence SMV can not verify any of them in 24 hours. The verification time for scr1 is 834 seconds, and for prop5 is 8418 seconds, which are worse than our results.

We compare the systems we built with and without the redundancy removal techniques described in this paper. In Table 1, the fifth to seventh columns are the results obtained without our techniques; while the last four 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. The last column is the number of redundant predicates our method is able to identify. 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. With the new method, the number of refinement iterations is usually smaller. We can usually identify a significant number of predicates as redundant. As a result, the number of predicates in the final abstraction is usually small.

circuit	# regs	# gates	ctrex	Old			New			
			length	time	iters	pred	time	iters	pred	red
scr1	243	2295	16	637.5	103	40	386.4	67	34	23
prop5	250	2342	17	2262.0	131	48	756.2	101	44	26
prop8	244	2304	true	288.5	68	35	159.8	40	25	20
prop9	244	2304	true	2448.7	146	46	202.7	43	27	25
prop10	244	2304	true	6229.3	161	55	178.2	50	25	23
prop12	247	2317	true	707.0	111	45	591.2	80	38	26

Table 1. Comparison without and with redundancy removal

11 Conclusion and Future Work

We have presented new algorithms for the identification and removal of redundant predicates. These algorithms enable us to identify three conditions for redundancy removal: equivalence induced redundancy, redundancy that preserves safety properties and redundancy that preserves bisimulation equivalence. Once a redundant predicate has been identified, a reduced abstract model can be efficiently computed without referring to the concrete model. Experimental results demonstrate the usefulness of our algorithms.

An interesting extension of the work presented in this paper is to identify redundant *sets of predicates*. That is, instead of identifying redundant predicates one at a time, sets of redundant predicates are identified together. In this setting the redundancy criteria may have to be different. The presented algorithms work for both hardware and software verification provided that the abstract model is finite state. The SLAM project uses *boolean programs* as the abstract model, which might have infinite control. It will be interesting to investigate how to extend our algorithm to handle boolean programs.

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 Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427, pages 319–331, Vancouver, Canada, 1998. Springer-Verlag.
4. Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science and Applied Mathematics Series. Academic Press, New York, NY, 1973.
6. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proc. of the 19th Annual Symposium on Principles of Programming Languages*, pages 343–354, 1992.
7. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *Twlfth Conference on Computer Aided Verification (CAV'00)*. Springer-Verlag, July 2000.
8. 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.
9. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
10. 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.
11. Michael Colon and Tomas E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, pages 293–304, 1998.
12. Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171, 1999.
13. Marcelo Glusman, Gila Kamhi, Sela Mador-Haim, Ranan Fraer, and Moshe Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *TACAS'03*, 2003.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages 58–70, 2002.
15. Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *Design Automation Conference*, pages 266–271, 1993.
16. Yassine Lachnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, pages 98–112, 2001.
17. 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.
18. K.L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS'03*, 2003.
19. Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *12th Conference on Computer Aided Verification*, number 1855 in LNCS, 2000.
20. Corina Pasareanu, Matthew Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted java programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, 2001.
21. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
22. H. Saidi and N. Shankar. Abstract and model check while you prove. In *11th Conference on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 443–454, July 1999.
23. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of ICCAD'01*, November 2001.