Predicate Abstraction with Minimum Predicates

⋆

Sagar Chaki Edmund Clarke Alex Groce Ofer Strichman

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA
chaki|emc|agroce|ofers@cs.cmu.edu

Abstract. Predicate abstraction is a popular abstraction technique employed in formal software verification. A crucial requirement to make predicate abstraction effective is to use as few predicates as possible, since the abstraction process is in the worst case exponential (in both time and memory requirements) in the number of predicates involved. If a property can be proven to hold or not hold based on a given finite set of predicates $\mathcal{P}$, the procedure we propose in this paper finds automatically a minimal subset of $\mathcal{P}$ that is sufficient for the proof. We explain how our technique can be used for more efficient verification of C programs. Our experiments show that predicate minimization can result in a significant reduction of both verification time and memory usage compared to earlier methods.

1 Introduction

Predicate abstraction [13] is a commonly used abstraction technique in formal verification of both software and hardware. Like other abstractions, when successful it can be used to prove the correctness (or incorrectness) of a property with only partial information about the reachable states of the system. This facilitates the verification of systems larger than would otherwise be possible. Predicate abstraction has been used widely both for hardware [5] and software [2, 9] verification. In this article we focus on its application to the verification of C programs.

Verification of programs typically concentrates on the control flow of the program (e.g. checking if a particular control point is reachable), rather than on the data manipulated by it (e.g. checking functional correctness). Predicate abstraction is a common abstraction technique used in this context. Given a program Π and a set of predicates $\mathcal{P}$, verification with predicate abstraction consists of constructing and analyzing an automaton $\mathcal{A}(\Pi, \mathcal{P})$, an abstraction of Π relative to $\mathcal{P}$.

We will describe in more detail predicate abstraction for verification of C programs in section 2. For now let us just mention that the process of constructing $\mathcal{A}(\Pi, \mathcal{P})$ is in the worst case exponential, both in time and space, in $|\mathcal{P}|$.

⋆ This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796.

Therefore a crucial point in deriving efficient algorithms based on predicate abstraction is the choice of a *small* set of predicates. In other words, one of the main challenges in making predicate abstraction effective is distinguishing a small set of predicates that are sufficient for determining whether a property holds or not. In this article we present an automated technique for finding the minimal such set from a given set of candidate predicates.

In the original article describing predicate abstraction [13] the process of selecting predicates is done manually. An automatic method for choosing predicates was suggested by Ball and Rajamani [2]. They follow a CounterExample Guided Abstraction Refinement (CEGAR) loop, which we now describe. Let ϕ be the property that we wish to verify over the program Π. We denote by $\mathcal{MC}$ a model checking algorithm that takes both $\mathcal{A}(\Pi, \mathcal{P})$ and ϕ as inputs and outputs TRUE if $\mathcal{A}(\Pi, \mathcal{P}) \models \phi$ and a counterexample τ otherwise. We assume ϕ is a safety property, so that τ is a finite acyclic trace of $\mathcal{A}(\Pi, \mathcal{P})$. Since τ is a trace of $\mathcal{A}(\Pi, \mathcal{P})$, it is often called an *abstract* trace. Let γ be a *trace concretization* function that maps every abstract trace to a sequence of instructions of Π consistent with the control flow graph. In order to check whether this sequence is a valid trace of Π, we define a *Trace Checking* algorithm $\mathcal{TC}$ that takes Π and τ as inputs and returns TRUE if $\gamma(\tau)$ is a valid trace of Π and FALSE otherwise. In the latter case τ is called a *spurious counterexample*. Finally, if τ is spurious, we need to eliminate it from the abstract model. We say that a set of predicates $\mathcal{P}'$ *eliminates* τ iff for every trace τ' of $\mathcal{A}(\Pi, \mathcal{P}')$, $\gamma(\tau) \neq \gamma(\tau')$; i.e. , the concretization of all traces in $\mathcal{A}(\Pi, \mathcal{P}')$ are different from $\gamma(\tau)$. Given these definitions, we now describe the four steps of the CEGAR loop (usually $\mathcal{P} = \emptyset$ initially):

1. **Abstract.** Construct $\mathcal{A}(\Pi, \mathcal{P})$.
2. **Verify.** If $\mathcal{MC}(\mathcal{A}(\Pi, \mathcal{P}), \phi) =$ TRUE, return *property holds.* Otherwise let τ be the counterexample.
3. **Check.** If $\mathcal{TC}(\Pi, \tau) =$ TRUE return *property does not hold.*
4. **Refine.** Update $\mathcal{P}$ so as to eliminate τ. Go to step 1.

Step 4 is the crucial one, and also the subject of this article. In previous work [2, 9] the refinement is done by adding predicates that eliminate the new spurious counterexample while maintaining the predicates that were found in previous iterations. This guarantees that no spurious counterexample will be repeated. However, this accumulative approach cannot guarantee a minimal set of predicates, because it depends on the order in which the counterexamples are identified and the choice of predicates at each step.

For example, consider a scenario where the first counterexample, τ_1, can be eliminated by either p_1 or p_2, and the process chooses p_1. Now it finds another counterexample, τ_2, which can only be eliminated by the predicate p_2. The process now proceeds with both p_1 and p_2, although p_2 by itself is sufficient to eliminate both τ_1 and τ_2. The framework that we present in this article, on the other hand, finds a minimal set of predicates that eliminate all the spurious counterexamples discovered so far. This guarantees a minimal set of predicates throughout the process, which is expected to reduce the overall verification time and required space. Our experimental results show that indeed the number of predicates and consequently the amount of memory required are significantly reduced.

Related work. Predicate abstraction was introduced by Graf and Saidi in [13]. It was subsequently used with considerable success in both hardware and software verification [2, 8, 9]. The notion of CEGAR was originally introduced by Kurshan [10] (originally termed *localization*) for model checking of finite state models. Both the abstraction and refinement techniques for such systems, as applied in his and consequent works, are essentially different than the predicate abstraction approach we follow. For example, abstraction in localization reduction is done by assigning non-deterministic values to selected sets of variables, while refinement corresponds to gradually returning to the original definition of these variables. More recently the CEGAR framework has also been successfully adapted for verifying infinite state systems [12], and in particular software [3, 9]. The problem of finding small sets of predicates (yet not minimal) is also being investigated in the context of hardware designs in [5].

The rest of this article is structured as follows. In the next section we discuss in more detail the CEGAR loop for predicate abstraction and how it is used for verifying C programs. In section 3 we describe in detail the procedure for selecting a minimal set of predicates. In section 4 we present the results of applying our technique to several realistic examples and detail our conclusions.

2 Predicate Abstraction/Refinement for C Programs

In the introduction we discussed the overall structure of a CEGAR loop. In this section we explain how this framework can be applied for verifying C programs. We do so by describing how the various basic blocks of the CEGAR loop are implemented. In particular, we discuss the construction of $\mathcal{A}(\Pi, \mathcal{P})$ in section 2.1, the notion of trace concretization (γ) in section 2.2, the trace checking algorithm $\mathcal{TC}$ in section 2.3, and a method for checking whether a set of predicates eliminates a spurious counterexample in section 2.4.

2.1 Constructing the abstract model

We begin with the process of constructing $\mathcal{A}(\Pi, \mathcal{P})$ given a C program Π and an initial set of predicates $\mathcal{P}$. For the sake of simplicity, we assume that Π consists of a single monolithic C *main* procedure obtained via inlining (we disallow function pointers and recursion in order to make inlining effective). Without loss of generality, we can assume that there are only four kinds of statements in Π: assignments, `if-then-else` branches, `goto` and `return`. We denote by *Stmt* the set of statements of Π and by *Exp* the set of all *pure* (side-effect free) C expressions over the variables of Π. As a running example we use the following simple C program and the property that label `L4` is unreachable.

```
    int x,y;
L0: x = 1;
L1: y = 1;
L2: if (x == y)
L3:     y = 1;
L4: else y = 2;
```

Initial abstraction with control flow automata. The construction of $\mathcal{A}(\Pi, \mathcal{P})$ begins with the construction of the control flow automaton (CFA) of Π. The states of a CFA correspond to control points in the program. The transitions between states in the CFA correspond to possible transitions between their associated control points in the program, *assuming that every branch in the program can be taken*. Thus, a CFA of a program is a conservative abstraction of the program's control flow, i.e. it allows a superset of the possible traces of the program.

Formally the CFA is a 4-tuple $\langle S_{CF}, I_{CF}, T_{CF}, \mathcal{L} \rangle$ where:

- S_{CF} is a set of states.
- $I_{CF} \in S_{CF}$ is an initial state.
- $T_{CF} \subseteq S_{CF} \times S_{CF}$ is a set of transitions.
- $\mathcal{L} : S_{CF} \setminus \{\mathit{final}\} \rightarrow \mathit{Stmt}$ is a labeling function.

S_{CF} contains a distinguished *final* state which does not belong to the domain of $\mathcal{L}$. The transitions between states reflect the flow of control between their labeling statements: $\mathcal{L}(I_{CF})$ is the initial statement of Π and $(s_1, s_2) \in T_{CF}$ iff one of the following conditions hold:

- $\mathcal{L}(s_1)$ is an assignment or `goto` with $\mathcal{L}(s_2)$ as its unique successor.
- $\mathcal{L}(s_1)$ is a branch with $\mathcal{L}(s_2)$ as its `then` or `else` successor.
- $\mathcal{L}(s_1)$ is a `return` statement and $s_2 = \mathit{final}$.

The CFA is equivalent, as we will shortly see, to $\mathcal{A}(\Pi, \emptyset)$.

Example 1. The CFA of our example program is shown in Figure 1(a), where every state s is labeled with $\mathcal{L}(s)$. Henceforth we will refer to each CFA state by the corresponding statement label. We will use *final* for the final state. Therefore the states of the CFA in Figure 1(a) are `L0 ...L4` and *final* with `L0` being the initial state. □

Predicate inference. The main challenge in predicate abstraction is to identify the predicates that are necessary for proving the given property. In our framework we require $\mathcal{P}$ to be a subset of the branch statements in Π. Therefore we sometimes refer to $\mathcal{P}$ or subsets of $\mathcal{P}$ simply as a set of branches, where the actual meaning is the predicates that serve as the guards in these branches. The construction of $\mathcal{A}(\Pi, \mathcal{P})$ associates with each state s of the CFA a finite subset of *Exp* derived from $\mathcal{P}$, denoted by $\mathcal{P}_s$. The process of constructing the $\mathcal{P}_s$'s from $\mathcal{P}$ is known as *predicate inference* and is described by the algorithm *PredInfer* in Figure 2. Note that $\mathcal{P}_s$ is always $\emptyset$ if s is either the *final* state or $\mathcal{L}(s)$ is a `return` statement.

The algorithm uses a procedure for computing the *weakest precondition* $\mathcal{WP}$ of a predicate p relative to a given statement. We define $\mathcal{WP}$ in the same way as

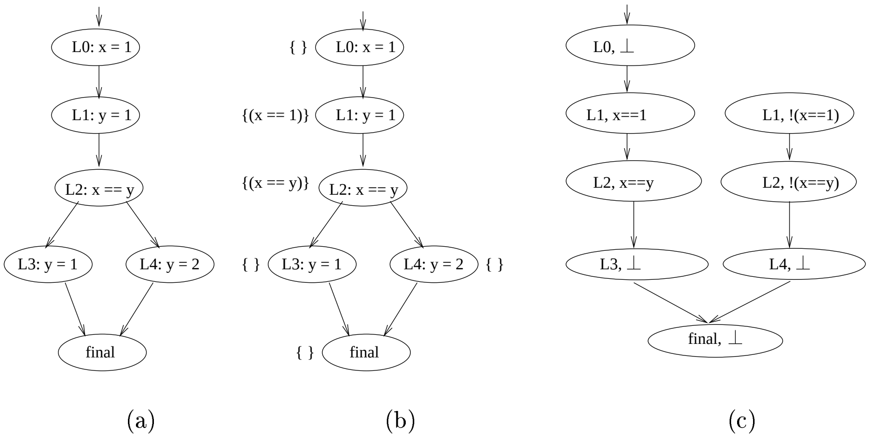

Fig. 1. (a) The CFA for our example program, (b) The CFA labeled with inferred predicates if $\mathcal{P} = \{(x == y)\}$, i.e., it contains the only branch in the program, and (c) The abstract automaton $\mathcal{A}(\Pi, \mathcal{P})$, which proves that `L4` is not reachable.

Ball and Rajamani [2]. First, consider a C assignment statement a of the form $v = e$;. Let φ be a pure C expression ($\varphi \in Exp$). Then the weakest precondition of φ with respect to a, denoted by $\mathcal{WP}(\varphi, a)$ is obtained from φ by replacing every occurrence of v in φ with e. A second case considers a C assignment statement a in which e is assigned to a variable whose address is stored in v, i.e. a is of the form $*v = e$;. Let $\{v_1, \ldots, v_n\}$ be the set of variables appearing in φ and for $1 \le i \le n$ let a_i be the assignment statement $v_i = e$; $\mathcal{WP}(\varphi, a)$ is then: $(||_{i=1}^{n}((v == \&v_i)\ \&\&\ \mathcal{WP}(\varphi, a_i)))\ ||\ (\&\&_{i=1}^{n}((v! = \&v_i))\ \&\&\ \varphi)$

```
Input: Set of branch statements 𝒫
Output: Set of 𝒫_s's associated with each CFA state
Initialize: ∀s ∈ S_CF, 𝒫_s := ∅
Forever do
  For each s ∈ S_CF do
    If ℒ(s) is an assignment statement and ℒ(s') is its successor
      For each p' ∈ 𝒫_s' add 𝒲𝒫(p', ℒ(s)) to 𝒫_s
    Else if ℒ(s) is a branch statement with condition c
      If ℒ(s) ∈ 𝒫 add c to 𝒫_s
      If ℒ(s') is a 'then' or 'else' successor of ℒ(s), 𝒫_s := 𝒫_s ∪ 𝒫_s'
    Else If ℒ(s) is a 'goto' statement with successor ℒ(s'), 𝒫_s := 𝒫_s ∪ 𝒫_s'
  If no 𝒫_s was modified in the 'for' loop, exit
```

Fig. 2. Algorithm *PredInfer* for predicate inference.

The weakest precondition is clearly an element of *Exp* as well. The purpose of predicate inference is to create $\mathcal{P}_s$'s that lead to a very precise abstraction of the program relative to the predicates in $\mathcal{P}$. Intuitively, this is how it works. Let $s, t \in S_{CF}$ such that $\mathcal{L}(s)$ is an assignment statement and $(s, t) \in T_{CF}$. Suppose a predicate p_t gets inserted in $\mathcal{P}_t$ at some point during the execution of *PredInfer* and suppose $p_s = \mathcal{WP}(p_t, \mathcal{L}(s))$. Now consider any execution state of Π where the control has reached $\mathcal{L}(t)$ after the execution of $\mathcal{L}(s)$. It is obvious that p_t will be true in this state iff p_s was true before the execution of $\mathcal{L}(s)$. In terms of the CFA, this means that the value of p_t after a transition from s to t can be determined precisely on the basis of the value of p_s before the transition. This motivates the inclusion of p_s in $\mathcal{P}_s$. The cases in which $\mathcal{L}(s)$ is not an assignment statement can be explained analogously.

Note that *PredInfer* may not terminate in the presence of loops in the CFA. However, this does not mean that our approach is incapable of handling C programs containing loops. In practice, we force termination of *PredInfer* by limiting the maximum size of any $\mathcal{P}_s$. Using the resulting $\mathcal{P}_s$'s, we can compute the states and transitions of the abstract model as described in the next section. Irrespective of whether *PredInfer* was terminated forcefully or not, the resulting model is guaranteed to be a sound abstraction of Π. We have found this approach to be very effective in practice. A similar algorithm was proposed by Dams and Namjoshi [7].

Example 2. Consider the CFA described in Example 1. Suppose $\mathcal{P}$ contains the only branch (`L2`) in our example program. Then *PredInfer* begins with $\mathcal{P}_{\texttt{L2}} = \{(x == y)\}$. From this it obtains $\mathcal{P}_{\texttt{L1}} = \{\mathcal{WP}((x == y), y = 1;)\} = \{(x == 1)\}$ and then $\mathcal{P}_{\texttt{L0}} = \{\mathcal{WP}((x == 1), x = 1;)\} = \{(1 == 1)\}$. As $(1 == 1)$ is trivially true, we do not include it in $\mathcal{P}_{\texttt{L0}}$. Thus $\mathcal{P}_{\texttt{L0}} = \emptyset$. Finally $\mathcal{P}_{\texttt{L3}} = \mathcal{P}_{\texttt{L4}} = \mathcal{P}_{final} = \emptyset$. Figure 1(b) shows the CFA with each state s labeled on the outside by $\mathcal{P}_s$. $\square$

The states and transitions of the abstract model. So far we have described a method for computing the initial abstraction (the CFA) and a set of predicates associated with each location in the program. The states of the abstract system $\mathcal{A}(\Pi, \mathcal{P})$ correspond to the various possible valuations of the predicates in each location (this is the reason why the abstract graph is exponential in the number of predicates). Formally, for a CFA node s suppose $\mathcal{P}_s = \{p_1, \ldots, p_k\}$. Then a *valuation* of $\mathcal{P}_s$ is a Boolean vector $v_1, \ldots, v_k$. Let $\mathcal{V}_s$ be the set of all predicate valuations of $\mathcal{P}_s$. Then the *predicate concretization* function $\Gamma_s : \mathcal{V}_s \to Exp$ is defined as follows. Given a valuation $V = \{v_1, \ldots, v_k\} \in \mathcal{V}_s$, $\Gamma_s(V) = \bigwedge_{i=1}^{k} p_i^{v_i}$ where $p_i^{\mathrm{TRUE}} = p_i$ and $p_i^{\mathrm{FALSE}} = \neg p_i$. As a special case, if $\mathcal{P}_s = \emptyset$, then $\mathcal{V}_s = \{\perp\}$ and $\Gamma_s(\perp) = \mathrm{TRUE}$.

Example 3. Suppose $\mathcal{P}_s = \{(a == 0), (b > 5), (c < d)\}$, $V_1 = \{0, 1, 1\}$ and $V_2 = \{1, 0, 1\}$. Then $\Gamma_s(V_1) = (\neg(a == 0)) \wedge (b > 5) \wedge (c < d)$ and $\Gamma_s(V_2) = (a == 0) \wedge (\neg(b > 5)) \wedge (c < d)$. $\square$

Computing the transitions between the states in $\mathcal{A}(\Pi, \mathcal{P})$ requires a theorem prover. We add a transition between two abstract states unless we can prove that there is no transition between their corresponding concrete states. If we cannot prove this, we say that the two states (or the two formulas representing them) are *admissible*. This problem can be reduced to the problem of deciding whether $\neg(\psi_1 \wedge \psi_2)$ is valid, where ψ_1 and ψ_2 are arbitrary quantifier free first order logic formulas. In general this problem is known to be undecidable. However for our purposes it is sufficient that the theorem prover be sound and always terminate. Several publicly available theorem provers (such as Simplify [11]) have this characteristic.

Given arbitrary formulas ψ_1 and ψ_2, we say that the formulas are *admissible* if the theorem prover returns FALSE or UNKNOWN on $\neg(\psi_1 \wedge \psi_2)$. We denote this by $Adm(\psi_1, \psi_2)$. Otherwise the formulas are inadmissible, denoted by $\neg Adm(\psi_1, \psi_2)$.

A procedure for constructing $\mathcal{A}(\Pi, \mathcal{P})$. We now define $\mathcal{A}(\Pi, \mathcal{P})$. Formally, it is a triple $\langle S_{\mathcal{A}}, I_{\mathcal{A}}, T_{\mathcal{A}}\rangle$ where:
- $S_{\mathcal{A}} = \cup_{s \in S_{CF}} \{s\} \times \mathcal{V}_s$ is the set of states.
- $I_{\mathcal{A}} = \{I_{CF}\} \times \mathcal{V}_{I_{CF}}$ is the initial set of states.
- $T_{\mathcal{A}} \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ is the transition relation, defined as follows: $((s_1, V_1), (s_2, V_2)) \in T_{\mathcal{A}}$ iff $(s_1, s_2) \in T_{CF}$ and one of the following conditions hold:
  1. $\mathcal{L}(s_1)$ is an assignment statement and $Adm(\Gamma_{s_1}(V_1), \mathcal{WP}(\Gamma_{s_2}(V_2), \mathcal{L}(s_1)))$.
  2. $\mathcal{L}(s_1)$ is a branch statement with a branch condition c, $\mathcal{L}(s_2)$ is its `then` successor, $Adm(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$ and $Adm(\Gamma_{s_1}(V_1), c)$.
  3. $\mathcal{L}(s_1)$ is a branch statement with a branch condition c, $\mathcal{L}(s_2)$ is its `else` successor, $Adm(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$ and $Adm(\Gamma_{s_1}(V_1), \neg c)$.
  4. $\mathcal{L}(s_1)$ is a `goto` statement and $Adm(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$.
  5. $\mathcal{L}(s_1)$ is a `return` statement and s_2 is the *final* state.

Example 4. Recall the CFA from Example 1 and the predicates corresponding to CFA nodes discussed in Example 2. The $\mathcal{A}(\Pi, \mathcal{P})$ obtained in this case appears in Figure 1(c). Let us see why there is a transition from $(\mathtt{L0}, \bot)$ to ($\mathtt{L1}$, TRUE). Since $\mathcal{L}(\mathtt{L0})$ is an assignment statement, by rule 1 above we compute the following expressions:
- $\Gamma_{\mathtt{L0}}(\bot) =$ TRUE
- $\Gamma_{\mathtt{L1}}(\text{TRUE}) = (x == 1)$.
- $\mathcal{L}(\mathtt{L0}) = (x = 1)$
- $\mathcal{WP}(\Gamma_{\mathtt{L1}}(\text{TRUE}), \mathcal{L}(\mathtt{L0})) = \mathcal{WP}((x == 1), x = 1;) = (1 == 1) =$ TRUE
- Adm(TRUE,TRUE).

Thus, we add a transition from $(\mathtt{L0}, \bot)$ to ($\mathtt{L1}$, TRUE). Examining a possible transition from $(\mathtt{L0}, \bot)$ to ($\mathtt{L1}$, FALSE), we similarly compute $\Gamma_{\mathtt{L1}}(\text{FALSE}) = (\neg(x == 1))$ and $\mathcal{WP}((\neg(x == 1)), x = 1;) = (\neg(1 == 1))$. Since $\neg Adm(\text{TRUE}, (\neg(1 == 1)))$, there is no transition between these two abstract states. The presence or absence of other transitions can be explained in a similar manner. As no state labeled by $\mathtt{L4}$ is reachable, we have proven that our example property holds. □

Clearly, if we do not limit the size of $\mathcal{P}_s$, $|S_{\mathcal{A}}|$ is exponential in $|\mathcal{P}|$. Hence so are the worst case space and time complexities of constructing $\mathcal{A}(\Pi, \mathcal{P})$.

2.2 Trace concretization

A trace of $\mathcal{A}(\Pi, \mathcal{P})$ is a finite sequence $\langle (s_1, V_1), \ldots, (s_n, V_n)\rangle$ such that (i) for $1 \leq i \leq n$, $(s_i, V_i) \in S_{\mathcal{A}}$, (ii) $(s_1, V_1) \in I_{\mathcal{A}}$ and (iii) for $1 \leq i < n$,

```
Input: A trace τ of A(Π,P) s.t. γ(τ) = ⟨s_1,...,s_n⟩
Output: TRUE iff τ is valid (can be simulated on the concrete system)
Variable: X of type formula
Initialize: X := TRUE
For i = n to 1
  If s_i is an assignment
   X := WP(X, s_i)
  Else If s_i is a branch with condition c
   If (i < n)
    If s_{i+1} is the 'then' successor of s_i, X := X ∧ c
    else X := X ∧ ¬c
  If (X ≡ FALSE) return FALSE
Return TRUE
```

Fig. 3. Algorithm $\mathcal{TC}$ to check the validity of a trace of Π.

$((s_i, V_i), (s_{i+1}, V_{i+1})) \in T_{\mathcal{A}}$. Given such a trace $\tau = \langle (s_1, V_1), \ldots, (s_n, V_n) \rangle$ of $\mathcal{A}(\Pi, \mathcal{P})$, the concretization of τ is defined as $\gamma(\tau) = \langle \mathcal{L}(s_1), \ldots, \mathcal{L}(s_n) \rangle$. Thus, the concretization of an abstract trace is a trace of Π: a sequence of statements that correspond to some trace in the control flow graph of Π.

2.3 Trace checking

The $\mathcal{TC}$ algorithm, described in Figure 3, takes Π and a counterexample τ as inputs and returns TRUE if $\gamma(\tau)$ is a valid trace of Π. This is a backward traversal based algorithm. There is an equivalent algorithm [3] that is forward traversal based and uses strongest postconditions instead of weakest preconditions.

2.4 Checking trace elimination

Given a spurious counterexample $\tau = \langle (s_1, V_1), \ldots, (s_n, V_n) \rangle$ and a set of branches $\mathcal{P}$, we will need to determine if $\mathcal{P}$ eliminates τ. To do so we: (i) construct $\mathcal{A}(\Pi, \mathcal{P})$ and (ii) determine if there exists a trace τ' of $\mathcal{A}(\Pi, \mathcal{P})$ such that $\gamma(\tau) = \gamma(\tau')$. The algorithm, called *TraceEliminate*, is described in Figure 4[1].

3 Predicate Minimization

We now present the algorithm for discovering a *minimal* set of branches $\mathcal{P}$ of a program π that will help us prove or disprove a safety property ϕ.

3.1 The *Sample-and-Eliminate* algorithm

Algorithm *Sample-and-Eliminate*, described in Figure 5, is based on an abstraction refinement loop that keeps the set of predicates minimal throughout the process. It is modeled after the *Sample-and-Separate* algorithm [6], where it is

[1] Note that in practice this step can be carried out in an on-the-fly manner without constructing the full $\mathcal{A}(\Pi, \mathcal{P})$.

```
Input: Spurious trace τ s.t. γ(τ) = ⟨s_1, ..., s_n⟩ and a set of predicates 𝒫
Output: TRUE if τ is eliminated by 𝒫 and FALSE otherwise
Compute 𝒜(Π, 𝒫) = ⟨S_𝒜, I_𝒜, T_𝒜⟩
Variable: X, Y of type subset of S_𝒜
Initialize: X := {(s, V) ∈ S_𝒜 | s = s_1}
If (X = ∅) return TRUE
For i = 2 to n do
   Y := {(s', V') ∈ S_𝒜 | (s' = s_i) ∧ ∃(s, V) ∈ X . ((s, V), (s', V')) ∈ T_𝒜}
   If (Y = ∅) return TRUE
   X := Y
Return FALSE
```

Fig. 4. Algorithm *TraceEliminate* to check if a spurious trace can be eliminated.

used in a CEGAR framework for hardware verification. At each step it finds a counterexample if one exists and checks whether it corresponds to a concrete counterexample, as usual. Unlike previous approaches [3, 9], however, it finds a minimal set of predicates that eliminate all the concrete spurious traces that were found so far (in the last line of the loop.) Our approach to solving this minimization problem is the subject of Section 3.2.

```
Input: Program Π, safety property φ
Output: TRUE if proved that Π ⊨ φ, FALSE if proved Π ⊭ φ, and UNKNOWN
otherwise.
Variable: T set of spurious counterexamples, P set of predicates
Initialize: T := ∅, P := ∅
Forever do
   If ℳ𝒞(𝒜(Π, P), φ) = TRUE return TRUE
   Else let τ be the abstract counterexample
   If 𝒯𝒞(τ) = TRUE return FALSE
   If P is the set of all branches in Π then return UNKNOWN
   T := T ∪ {τ}
   P := minimal set of branches of Π that eliminates all elements of T
```

Fig. 5. Algorithm *Sample-and-Eliminate* uses a minimal set of predicates taken from a program's branches to prove or disprove $\Pi \models \phi$, if such a proof is possible.

3.2 Minimizing the eliminating set

The last line of *Sample-and-Eliminate* presents the following problem: given a set of spurious counterexamples T and a set of candidate predicates P (all the branches of Π in our case), find a minimal set $p \subseteq P$ which eliminates all the traces in T. We present a three step algorithm for solving this problem. **First**, find a mapping $T \mapsto 2^{2^P}$ between each trace in T and the set of sets of predicates in P that eliminate it. This can be achieved by iterating through every $p \subseteq P$ and $\tau \in T$, using *TraceEliminate* to determine if p can eliminate τ. This approach is exponential in $|P|$ but below we list several ways to reduce the number of attempted combinations:

- Limit the size or number of attempted combinations to a small constant, e.g. 5, assuming that most traces can be eliminated by a small set of predicates.
- Stop after reaching a certain size of combinations if any eliminating solutions have been found.
- Break up the control flow graph into blocks and only consider combinations of predicates within blocks (keeping combinations in other blocks fixed).
- Use data flow analysis to only consider combinations of related predicates.
- For any $\tau \in T$, if a set p eliminates τ, ignore all supersets of p with respect to τ (as we are seeking a minimal solution).

Second, encode each predicate $p_i \in P$ with a new Boolean variable p_i^b. We use the terms 'predicate' and 'the Boolean encoding of the predicate' interchangeably. **Third**, derive a Boolean formula σ, based on the predicate encoding, that represents all the possible combinations of predicates that eliminate the elements of T. We use the following notation in the description of σ. Let $\tau \in T$ be a trace:

- k_τ denotes the number of sets of predicates that eliminate τ $(1 \leq k_\tau \leq 2^{|P|})$.
- $s(\tau, i)$ denotes the i-th set $(1 \leq i \leq k_\tau)$ of predicates that eliminates τ. We use the same notation for the conjunction of the predicates in this set.

The formula σ is defined as follows:

$$\sigma \stackrel{\text{def}}{=} \bigwedge_{\tau \in T} \bigvee_{i=1}^{k_\tau} s(\tau, i) \tag{1}$$

For any satisfying assignment to σ, the predicates whose Boolean encodings are assigned TRUE are sufficient for eliminating all elements of T.

From the various possible satisfying assignments to σ, we look for the one with the smallest number of positive assignments. This assignment represents the minimal number of predicates that are sufficient for eliminating T. Since σ includes disjunctions, it cannot be solved directly with a 0-1 ILP solver. We therefore use PBS [1], a solver for Pseudo Boolean Formulas.

A pseudo-Boolean formula is of the form $\sum_{i=1}^{n} c_i \cdot b_i \bowtie k$, where b_i is a Boolean variable and c_i is a rational constant for $1 \leq i \leq n$. k is a rational constant and $\bowtie$ represents one of the inequality or equality relations $(\{<, \leq, >, \geq, =\})$. Each such constraint can be expanded to a CNF formula (hence the name pseudo-Boolean), but this expansion can be exponential in n. PBS does not perform this expansion, but rather uses an algorithm designed in the spirit of the Davis-Putnam-Loveland algorithm that handles these constraints directly. PBS accepts as input standard CNF formulas augmented with pseudo-Boolean constraints. Given an objective function in the form of pseudo-Boolean formula, PBS finds an optimal solution by repeatedly tightening the constraint over the value of this function until it becomes unsatisfiable. That is, it first finds a satisfying solution and calculates the value of the objective function according to this solution. It then adds a constraint that the value of the objective function should be smaller by one. This process is repeated until the formula becomes unsatisfiable. The objective function in our case is to minimize the number of chosen predicates (by minimizing the number of variables that are assigned TRUE):

$$\min \sum_{i=1}^{n} p_i^b \qquad (2)$$

Example 5. Suppose that the trace τ_1 is eliminated by either $\{p_1, p_3, p_5\}$ or $\{p_2, p_5\}$ and that the trace τ_2 can be eliminated by either $\{p_2, p_3\}$ or $\{p_4\}$. The objective function is $\min \sum_{i=1}^{5} p_i^b$ and is subject to the constraint:

$$\begin{aligned}\sigma = \; & ((p_1^b \wedge p_3^b \wedge p_5^b) \vee (p_2^b \wedge p_5^b)) \wedge \\ & ((p_2^b \wedge p_3^b) \vee (p_4^b))\end{aligned}$$

The minimal satisfying assignment in this case is $p_2^b = p_5^b = p_4^b =$ TRUE. □

Other techniques for solving this optimization problem are possible, including minimal hitting sets and logic minimization. The PBS step, however, has not been a bottleneck in any of our experiments.

4 Experiments and Conclusions

We implemented our technique inside the MAGIC [4] tool. MAGIC was designed to check *weak simulation* of properties of labeled transition systems (LTSs) derived from C programs. We experimented with MAGIC with and without predicate optimization. We also performed experiments with a greedy predicate minimization strategy implemented on top of MAGIC. In each iteration, this strategy first adds predicates sufficient to eliminate the spurious counterexample to the predicate set $\mathcal{P}$. Then it attempts to reduce the size of the resulting $\mathcal{P}$ by using the algorithm described in Figure 6. The advantage of this approach is that it requires only a small overhead (polynomial) compared to *Sample-and-Eliminate*, but on the other hand it does not guarantee an optimal result. Further, we performed experiments with Berkeley's BLAST [9] tool. BLAST also takes C programs as input, and uses a variation of the standard CEGAR loop based on *lazy abstraction*, but without minimization. Lazy abstraction refines an abstract model while allowing different degrees of abstraction in different parts of a program, without requiring recomputation of the entire abstract model in each iteration. Laziness and predicate minimization are, for the most part, orthogonal techniques. In principle a combination of the two might produce better results than either in isolation.

Benchmarks. We used two kinds of benchmarks. A small set of relatively simple benchmarks were derived from the examples supplied with the BLAST distribution and regression tests for MAGIC. The difficult benchmarks were derived from the C source code of `openssl-0.9.6c`, several thousand lines of code implementing the SSL protocol used for secure transfer of information over the Internet. A critical component of this protocol is the initial *handshake* between

```
Input: Set of predicates 𝒫
Output: Subset of 𝒫 that eliminates all spurious counterexamples so far
Variable: X of type set of predicates
LOOP: Create a random ordering ⟨p_1, ..., p_k⟩ of 𝒫
For i = 1 to k do
   X := 𝒫 \ {p_i}
   If X can eliminate every spurious counterexample seen so far
     𝒫 := X
     Goto LOOP
Return 𝒫
```

Fig. 6. Greedy predicate minimization algorithm.

a server and a client. We verified different properties of the main routines that implement the handshake. The names of benchmarks that are derived from the server routine and client routine begin with `ssl-srvr` and `ssl-clnt` respectively. In all our benchmarks, the properties are satisfied by the implementation. The server and client routines have roughly 350 lines each but, as our results indicate, are non-trivial to verify.

Results. Figure 7 summarizes our results. Time for all experiments is given in seconds. All experiments were performed on an AMD Athlon XP 1600 machine with 900 MB of RAM running RedHat 7.1. The column Iter reports the number of iterations through the CEGAR loop necessary to complete the proof. Predicates are listed differently for the two tools. For BLAST, the first number is the total number of predicates discovered and used and the second number is the number of predicates active at any one point in the program (due to lazy abstraction this may be smaller). In order to force termination we imposed a limit of three hours on the running time. We denote by '*' in the Time column examples that could not be solved in this time limit. In these cases the other columns indicate relevant measurements made at the point of forceful termination.

For MAGIC, the first number is the total number of expressions used to prove the property, i.e. $|\cup_{s \in S_{CF}} \mathcal{P}_s|$. The number of predicates (the second number) may be smaller, as MAGIC combines multiple mutually exclusive expressions (e.g. $x == 1$, $x < 1$, and $x > 1$) into a single, possibly non-binary predicate, having a number of values equal to the number of expressions (plus one, if the expressions do not cover all possibilities.) The final number for MAGIC is the size of the final $\mathcal{P}$. For experiments in which memory usage was large enough to be a measure of state space size rather than overhead, we also report memory usage (in megabytes).

The first MAGIC results are for the MAGIC tool operating in the standard refinement manner: in each iteration, predicates sufficient to eliminate the spurious counterexample are added to the predicate set. The second MAGIC results are for predicate minimization. Rather than solving the full optimization problem, we simplified the problem as described in section 3. In particular, for each trace we only considered the first 1,000 combinations and only generated 20 eliminating combinations. The combinations were considered in increasing order

	BLAST				MAGIC				MAGIC + MINIMIZE			
Program	Time	Iter	Pred	Mem	Time	Iter	Pred	Mem	Time	Iter	Pred	Mem
funcall-nes	**1**	3	13/10	×	5	2	10/9/1	×	5	2	10/9/1	×
fun_lock	5	7	7/7	×	5	4	8/3/3	×	6	4	8/3/3	×
driver.c	**1**	4	3/2	×	6	5	6/2/4	×	5	5	6/2/4	×
read.c	6	11	20/11	×	5	2	15/5/2	×	5	2	15/5/1	×
socket-y-01	5	13	16/6	×	5	3	12/4/2	×	6	3	12/4/2	×
opttest.c	7499	38	37/37	231	**145**	5	7/7/8	63	247	25	4/4/4	63
ssl-srvr-1	2398	16	33/8	175	250	12	56/5/22	43	**226**	14	5/4/2	38
ssl-srvr-2	691	13	68/8	60	752	16	72/6/30	72	**216**	14	5/4/2	38
ssl-srvr-3	1162	14	32/7	103	331	12	56/5/22	47	**200**	12	5/4/2	38
ssl-srvr-4	284	11	27/5	44	677	14	63/6/26	72	**170**	9	5/4/2	38
ssl-srvr-5	1804	19	52/5	71	**71**	5	22/4/8	24	205	13	5/4/2	36
ssl-srvr-6	*	39	90/10	805	11840	23	105/11/44	1187	**359**	14	8/4/3	89
ssl-srvr-7	359	11	76/9	37	2575	20	94/7/38	192	**196**	11	5/4/2	38
ssl-srvr-8	*	25	35/5	266	**130**	8	32/5/14	58	211	10	8/4/3	40
ssl-srvr-9	337	10	76/9	36	2621	15	65/8/28	183	**316**	20	11/4/4	38
ssl-srvr-10	8289	20	35/8	148	561	16	75/6/30	73	**241**	14	8/4/3	38
ssl-srvr-11	547	11	78/11	51	4014	19	89/8/36	287	**356**	24	8/4/3	38
ssl-srvr-12	2434	21	80/8	120	7627	22	102/9/42	536	**301**	17	8/4/3	42
ssl-srvr-13	608	12	79/12	54	3127	17	75/9/32	498	**436**	29	11/4/4	38
ssl-srvr-14	10444	27	84/10	278	7317	22	102/9/42	721	**406**	20	8/4/3	52
ssl-srvr-15	*	31	38/5	436	615	15	81/28/5	188	**179**	7	8/4/3	40
ssl-srvr-16	*	33	87/10	480	3413	21	98/8/40	557	**356**	17	8/4/3	58
ssl-clnt-1	348	16	28/5	43	**110**	10	43/4/18	25	156	12	5/4/2	31
ssl-clnt-2	523	15	28/4	52	**156**	11	53/5/20	31	185	18	5/4/2	29
ssl-clnt-3	469	14	29/5	49	421	13	52/7/24	58	**195**	21	5/4/2	29
ssl-clnt-4	380	13	27/4	45	**125**	10	35/5/18	27	191	19	5/4/2	29
TOTAL	81794	447	1178 /221	3584	46904	322	1428/185 /559	4942	**5375**	356	191/107 /67	880
AVERAGE	3146	17	45/9	171	1804	12	55/7/22	235	**207**	14	7/4/3	42

Fig. 7. Results for BLAST and MAGIC with different refinement strategies. '*' indicate run-time longer than 3 hours. '×' indicate negligible values. Best results are emphasized.

of size. After all combinations of a particular size had been tried, we checked whether at least one eliminating combination had been found. If so, no further combinations were tried. In the smaller examples we observed no loss of optimality due to these restrictions. We also studied the effect of altering these restrictions on the larger benchmarks and we report on our findings later. Due to lack of space, we omit the measurements for the greedy approach.

For the smaller benchmarks, the various abstraction refinement strategies do not differ markedly. However, for our larger examples, taken from the SSL source code, the refinement strategy is of considerable importance. Predicate minimization, in general, reduced verification time (though there were a few exceptions to this rule, the average running time was considerably lower than for the other techniques, even with the cutoff on the running time). Moreover, predicate minimization reduced the memory needed for verification, which is an even more important bottleneck. Given that the memory was cutoff in some cases for other techniques before verification was complete, the results are even more compelling.

The greedy approach kept memory use fairly low, but almost always failed to find near-optimal predicate sets and converged much slower than the usual monotonic refinement or predicate minimization approaches. Further, it is not clear how much final memory usage would be improved by the greedy strategy if it were allowed to run to completion. In support of our conclusions, we present the "TOTAL" and "AVERAGE" measurements for the greedy approach. If these were presented in Figure 7, the TOTAL row would be: Time = 163163, Iter = 1775, Pred = 381/102/129 and Mem = 2182. Correspondingly, the AVERAGE row would be: Time = 6276, Iter = 68, Pred = 15/4/5 and Mem = 104.

Another major drawback of the greedy approach is its unpredictability. We observed that on any particular example, the greedy strategy might or might not complete within the time limit in different executions. Clearly, the order in which this strategy tries to eliminate predicates in each iteration is very critical to its success. Given that the strategy performs poorly on most of our benchmarks using a random ordering, more sophisticated ordering techniques may perform better. We leave this issue for future research.

		ssl-srvr-4						ssl-srvr-15						ssl-clnt-1					
ELM	SUB	Time	It	$\lvert\mathcal{P}\rvert$	Mem	T	G	Time	It	$\lvert\mathcal{P}\rvert$	Mem	T	G	Time	It	$\lvert\mathcal{P}\rvert$	Mem	T	G
50	250	656	8	2	64	34	1	1170	15	3	72	86	1	1089	13	2	67	66	1
100	250	656	8	2	64	34	1	1169	15	3	72	86	1	1089	13	2	67	66	1
150	250	657	8	2	64	34	1	1169	15	3	72	86	1	1090	13	2	67	66	1
200	250	656	8	2	64	34	1	1170	15	3	72	86	1	1089	13	2	67	66	1
250	250	656	8	2	64	34	1	1168	15	3	72	86	1	1090	13	2	67	66	1

Fig. 8. Results for optimality. ELM = MAXELM, SUB = MAXSUB, It is the number of iterations, T is the total number of eliminating subsets generated, and G is the maximum size of any eliminating subset generated.

Optimality. We experimented with two of the parameters that affect the optimality of our predicate minimization algorithm: (i) the maximum number of examined subsets (MAXSUB) and (ii) the maximum number of eliminating subsets generated (MAXELM) (that is, the procedure stops the search if MAXELM eliminating subsets were found, even if less than MAXSUB combinations were tried). We first kept MAXSUB fixed and took measurements for different values of MAXELM on a subset of our benchmarks viz. `ssl-srvr-4`, `ssl-srvr-15` and `ssl-clnt-1`. Our results, shown in Figure 8, clearly indicate that the optimality is practically unaffected by the value of MAXELM.

Next we experimented with different values of MAXSUB (the value of MAXELM was set equal to MAXSUB). The results we obtained are summarized in Figure 9. It appears that, at least for our benchmarks, increasing MAXSUB leads only to increased execution time without reduced memory consumption or number of predicates. The additional number of combinations attempted or constraints allowed does not lead to improved optimality. The most probable reason is that, as shown by our results, even though we are trying more combinations, the actual number or maximum size of eliminating combinations generated does not increase significantly. It would be interesting to investigate whether this is a feature of most real-life programs. If so, it would allow us, in most cases, to achieve near optimality by trying out only a small number of combinations or only combinations of small size.

	ssl-srvr-4					ssl-srvr-15					ssl-clnt-1				
SUB	Time	It	$\|\mathcal{P}\|$	Mem	T/M/G	Time	It	$\|\mathcal{P}\|$	Mem	T/M/G	Time	It	$\|\mathcal{P}\|$	Mem	T/M/G
100	262	8	2	44	34/2/1	396	12	3	50	62/2/1	310	11	2	40	58/2/1
200	474	7	2	57	27/2/1	917	14	3	65	81/2/1	683	12	2	51	63/2/1
400	1039	9	2	71	38/2/1	1110	8	3	76	45/2/1	2731	13	2	208	67/3/1
800	2182	7	2	165	25/2/1	2797	9	3	148	51/2/1	5843	14	2	296	75/3/1
1600	6718	9	2	410	35/3/1	10361	11	3	410	76/3/1	13169	12	2	633	61/3/1
3200	13656	9	2	461	40/3/1	14780	9	3	436	50/3/1	36155	12	2	1155	67/4/1
6400	26203	9	2	947	31/3/1	33781	10	3	792	51/3/1	> 57528	4	1	2110	22/4/1

Fig. 9. Results for optimality. SUB = MAXSUB, It is the number of iterations, T is the total number of eliminating subsets generated, M is the maximum size of subsets tried, and G is the maximum size of eliminating subsets generated.

Acknowledgments We thank Rupak Majumdar and Ranjit Jhala for their help with BLAST.

References

1. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo Boolean solver. In *Symposium on the theory and applications os satisfiability testing (SAT)*, pages 346–353, 2002.
2. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057, 2001.
3. T. Ball and S. K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, Redmond, January 2002.
4. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE), To appear*, 2003.
5. E. Clarke, O. Grumberg, M. Talupur, and D. Wang. Making predicate abstraction efficient: eliminating redundant predicates. In *To appear in CAV'03*.
6. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction - refinement using ILP and machine learning techniques. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, July 2002.
7. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI*, 2003.
8. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171, 1999.
9. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
10. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1995.
11. G. Nelson. *Techniques for Program Verification.* PhD thesis, Stanford, 1980.
12. V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. *Lecture Notes in Computer Science*, 1579:178–192, 1999.
13. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254, pages 72–83. Springer Verlag, 1997.