Localization and Register Sharing for Predicate Abstraction

Himanshu Jain[1,2], Franjo Ivančić[1], Aarti Gupta[1], and Malay K. Ganai[1]

[1] NEC Laboratories America, Inc., 4 Independence Way, Princeton, NJ 08540
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA

Abstract. In the domain of software verification, predicate abstraction has emerged to be a powerful and popular technique for extracting finite-state models from often complex source code. In this paper, we report on the application of three techniques for improving the performance of the predicate abstraction refinement loop. The first technique allows faster computation of the abstraction. Instead of maintaining a global set of predicates, we find predicates relevant to various basic blocks of the program by weakest pre-condition propagation along spurious program traces. The second technique enables faster model checking of the abstraction by reducing the number of state variables in the abstraction. This is done by re-using Boolean variables to represent different predicates in the abstraction in a safe way. However, some predicates are useful at many program locations and discovering them lazily in various parts of the program leads to a large number of abstraction refinement iterations. The third technique attempts to identify such predicates early in the abstraction refinement loop and handles them separately by introducing dedicated state variables for such predicates. We have incorporated these techniques into NEC's software verification tool F-Soft, and present promising experimental results for various case studies using these techniques.

1 Introduction

In the domain of software verification, *predicate abstraction* [2, 7, 9, 11] has emerged to be a powerful and popular technique for extracting finite-state models from often complex source code. It abstracts data by keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated.

The application of predicate abstraction to large programs depends crucially on the choice and usage of the predicates. If all predicates are tracked globally in the program, the analysis often becomes intractable due to the large number of predicate relationships. In Microsoft's SLAM [4] toolkit, this problem is handled by generating coarse abstractions using techniques such as *Cartesian approximation* and the *maximum cube length approximation* [3]. These techniques limit the number of predicates in each theorem prover query. The refinement of the abstraction is carried out by adding new predicates. If no new predicates are found, the spurious behavior is due to inexact predicate relationships. Such spurious behavior is removed by a separate refinement algorithm called CONSTRAIN [1].

The BLAST toolkit [13] introduced the notion of *lazy abstraction*, where the abstraction refinement is completely demand-driven to remove spurious behaviors. Recent work [14] describes a new refinement scheme based on interpolation [8], which adds new predicates to some program locations only, which we will call henceforth *localization of predicates*. On average the number of predicates tracked at each program location is small and thus, the localization of predicates enables predicate abstraction to scale to larger software programs.

This paper makes three novel contributions:

- Our first contribution is inspired by the lazy abstraction approach and the localization techniques implemented in BLAST. While BLAST makes use of interpolation, we use weakest pre-conditions to find predicates relevant at each program location. Given an infeasible trace $s_1;\ldots;s_k$, we find predicates whose values need to be tracked at each statement s_i in order to eliminate the infeasible trace. For any program location we only need to track the relationship between the predicates relevant at that location. Furthermore, since we use predicates based on weakest preconditions along infeasible traces, most of the predicate relationships are obtained from the refinement process itself. This enables us to significantly reduce the number of calls to back-end decision procedures leading to a much faster abstraction computation.
- The performance of BDD-based model checkers depends crucially on the number of state variables. Due to predicate localization most predicates are useful only in certain parts of the program. The state variables corresponding to these predicates can be *reused* to represent different predicates in other parts of the abstraction, resulting in a reduction of the total number of state variables needed. We call this *abstraction with register sharing*. This constitutes our second technique which reduces the number of state variables, enabling more efficient model checking of the abstract models.
- While the above techniques speed up the individual computations and the model checking runs of the abstractions, they might result in too many abstraction refinement iterations. This can happen if the value of a certain predicate needs to be tracked at multiple program locations, i.e., if the predicate is useful *globally* or at least in some large part of the program. Since we add predicates lazily only along infeasible traces, the fact that a predicate is globally useful for checking a property will be learned only through multiple abstraction refinement iterations. We make use of a simple heuristic for deciding when the value of a certain predicate may need to be tracked globally or in a complete functional scope. If the value of a predicate needs to be tracked in a large scope, then it is assigned a *dedicated* state variable which is not reused for representing the value of other predicates in the same scope.

Further Related Work: Rusu et al. [20] present a framework for proving safety properties that combines predicate abstraction, refinement using weakest pre-conditions and theorem proving. However, no localization of predicates is done in their work. Namjoshi et al. [18] use weakest pre-conditions for extracting finite state abstractions, from possibly infinite state programs. They compute the weakest pre-conditions starting from an initial set of predicates derived from the specification, control guards etc. This process is iterated until a fix-point is reached, or a user imposed bound on the number of iterations is reached. In the latter case, the abstraction might be too coarse to prove the given property. However, no automatic refinement procedure is described. The MAGIC tool [5] also makes use of weakest pre-conditions in a similar way. Both approaches have the disadvantage that the number of predicates tracked at each program location can be much higher, which may make the single model checking step difficult. In contrast, we propagate the weakest pre-conditions lazily, that is, only to the extent needed to remove infeasible traces.

In order to check if a sequence of statements in the C program is (in)feasible we use a SAT-solver as in [16]. The relationships between a set of predicates is found by making use of SAT-based predicate abstraction [6, 17]. We further improve the performance of SAT-based simulation of counterexamples and abstraction computation by making use of range analysis techniques [19, 21] to determine the maximum number of bits needed to represent each variable in the given program.

Outline: The following section describes the pre-processing of the source code with our software verification tool F-SOFT [15] and the localized abstraction refinement framework based on weakest pre-condition propagation. F-SOFT allows both SAT-based and BDD-based bounded and unbounded model checking of C and Java code. The third section presents an overview of the Boolean model builder that computes the abstraction with and without register sharing, while the fourth section describes our approach of dedicating abstract state variables to predicates. Section 5 discusses the experimental results, and we finish this paper with some concluding remarks.

2 A Localized Abstraction-Refinement Framework

2.1 Software Modeling

In this section, we briefly describe our software modeling approach that is centered around basic blocks as described in [15]. The preprocessing of the source code is performed before the abstraction refinement routine is invoked. A program counter variable is introduced to monitor progress in the control flow graph consisting of basic blocks. Our modeling framework allows *bounded recursion* through the introduction of a fixed depth function call stack, when necessary, and introduces special variables representing function return points for non-recursive functions. Due to space limitation, we omit the details of our handling of pointer variables, which can be found in [15]. It is based on adding simplified *pointer-free* assignments in the basic blocks.

2.2 Localization Information

The formula ϕ describes a set of program states, namely, the states in which the value of program variables satisfy ϕ. The *weakest pre-condition* [10] of a formula ϕ with respect to a statement s is the weakest formula whose truth before the execution of s entails the truth of ϕ after s terminates. We denote the weakest pre-condition of ϕ with respect to s by $WP(\phi, s)$. Let s be an assignment statement of the form `v = e;` and ϕ be a C expression. Then the weakest pre-condition of ϕ with respect to s, is obtained from ϕ by replacing every occurrence of v in ϕ with e.

Given an `if` statement with condition p, we write `assume p` or `assume ¬p`, depending upon the branch of the `if` statement that is executed. The weakest pre-condition of ϕ with respect to `assume p`, is given as $\phi \wedge p$. As mentioned earlier, pointer assignments are rewritten early on in our tool chain, thus allowing us to focus here on only the above cases. The weakest pre-condition operator is extended to a sequence of statements by $WP(\phi, s_1; s_2) = WP(WP(\phi, s_2), s_1)$. A sequence of statements $s_1; \ldots; s_k$ is said to be *infeasible*, if $WP(true, s_1; \ldots; s_k) = false$. Note that for ease of presentation, we present the following material using individual statements while the actual implementation uses a control flow graph consisting of basic blocks.

We define $child(s)$ to denote the set of statements reachable from s in one step in the control flow graph. Each statement s in the program keeps track of the following information: (1) A set of predicates denoted as $local(s)$ whose values need to be tracked before the execution of s. We say a predicate p is *active* at the statement s, if $p \in local(s)$. (2) A set of predicate pairs denoted as $transfer(s)$. Intuitively, if $(p_i, p_j) \in transfer(s)$, then the value of p_j after s terminates is equal to the value of p_i before the execution of s. Formally, a pair $(p_i, p_j) \in transfer(s)$ satisfies the following conditions:

- $p_i \in \{True, False\} \cup local(s)$.
- There exists $s' \in child(s)$, such that $p_j \in local(s')$.
- If s is an assignment statement, then $p_i = WP(p_j, s)$.
- If s is an assume statement, then $p_i = p_j$.

We refer to the sets $local(s)$ and $transfer(s)$ together as the *localization information* at the statement s. This information is generated during the refinement step, and is used for creating refined abstractions which eliminate infeasible traces.

Example: Consider the C program in Fig. 1(a) and the localization information shown in Fig. 1(d). Since $(p_4, p_3) \in transfer(s_1)$ and s_1 is an assignment, it means that $p_4(c = m)$ is the weakest pre-condition of $p_3(x = m)$ with respect to statement s_1. The value of predicate p_4 is useful only before the execution of s_1. After execution of s_1, predicate p_3 becomes useful.

2.3 Refinement using Weakest Pre-condition Propagation

Let $s_1; \ldots; s_k$ be an infeasible program trace. If s_i is of the form `assume` p_i, then the weakest pre-condition of p_i is propagated backwards from s_i until s_1. When computing the weakest pre-condition of a predicate p_i with respect to a statement s_j of the form `assume` p_j, we propagate the weakest pre-conditions of p_i and p_j separately. That is, we do not introduce a new predicate for $p_i \wedge p_j$. This is done to ensure that the predicates remain atomic. The *local* and the *transfer* sets for the various statements are updated during this process. The complete algorithm is given in Fig. 2.

Example: Consider the C program in Fig. 1(a) and an infeasible trace in Fig. 1(b). Assume that initially $local(s)$ and $transfer(s)$ sets are empty for each s. The refinement

```
s
1: x = c;
2: y = c + 1;
3: if (x == m);
4:    if (y != m+1);
5:      ERROR: ;
```

(a)

```
s
1: x = c;
2: y = c + 1;
3: assume (x == m);
4: assume (y != m+1);
```

(b)

s	local(s)	transfer(s)
1:	$\{p_2\}$	$\{(p_2,p_2)\}$
2:	$\{p_2\}$	$\{(p_2,p_1)\}$
3:	$\{p_1\}$	$\{(p_1,p_1)\}$
4:	$\{p_1\}$	

(c)

s	local(s)	transfer(s)
1:	$\{p_2,p_4\}$	$\{(p_2,p_2),(p_4,p_3)\}$
2:	$\{p_2,p_3\}$	$\{(p_2,p_1),(p_3,p_3)\}$
3:	$\{p_1,p_3\}$	$\{(p_1,p_1)\}$
4:	$\{p_1\}$	

(d)

Fig. 1. (a) A simple C program. (b) An infeasible program trace. (c) Status of $local(s)$ and $transfer(s)$ sets after the first iteration of the refinement algorithm (see Fig. 2). Predicates p_1, p_2 denote $y \neq m+1$ and $c \neq m$, respectively. (d) New additions to the $local(s)$ and $transfer(s)$ in the second iteration. p_3, p_4 denote $x = m$ and $c = m$ respectively.

algorithm in Fig. 2 is applied to the infeasible trace. The localization information after the first iteration ($i = 4$) and second iteration ($i = 3$) of the outer loop in the refinement algorithm, are shown in Fig. 1(c) and Fig. 1(d), respectively. No change occurs to the localization information for $i = 2$ and $i = 1$, since s_2 and s_1 do not correspond to `assume` statements.

If $s_1;\ldots;s_k$ is infeasible, then $WP(true, s_1;\ldots;s_k) = false$ by definition. Intuitively, the atomic predicates in $WP(true, s_1;\ldots;s_k)$ appear in $local(s_1)$. Thus, by finding the relationships between the predicates in $local(s_1)$, it is possible to construct a refined model which eliminates the infeasible trace. When an infeasible trace $s_1;\ldots;s_k$ is refined using the algorithm in Fig. 2, s_1 is stored into a set of statements denoted by *marked*. If a statement s is in the *marked* set, and the size of $local(s)$ is less than a certain threshold, then the abstraction routine computes the relationships between the predicates in $local(s)$ using SAT-based predicate abstraction [6, 17]. Otherwise, these relationships are determined lazily by detection of spurious abstract states [1].

Proof Based Analysis: The refinement algorithm described in Fig. 2 performs a backward weakest pre-condition propagation for each `assume` statement in the infeasible trace. However, neither all `assume` statements nor all assignments may be necessary for the infeasibility of the given trace. Propagating the weakest pre-conditions for all such statements will result in an unnecessary increase in the number of predicates active at each statement in the infeasible trace. We make use of the SAT-based proof of infeasibility of the given trace to determine the statements for which the weakest pre-condition propagation should be done [12]. Thus, the localization information is updated partially, in a way that is sufficient to remove the spurious behavior. The computation of an abstract model using the localization information is described in the next section.

```
Input: An infeasible trace $s_1;\ldots;s_k$
Algorithm:
for $i = k$ downto 1                                        //outer for loop
  if $s_i$ is of form (assume $\phi_i$) then                //propagate weakest pre-conditions
    $local(s_i) = local(s_i) \cup \{\phi_i\}$               //localize $\phi_i$ at $s_i$
    $seed = \phi_i$
    for $j = i-1$ downto 1                                  //inner for loop
      if $s_j$ is an assignment statement then
        $wp = WP(seed, s_j)$
      else
        $wp = seed$
      $local(s_j) = local(s_j) \cup \{wp\}$                 //localize $wp$ at $s_j$
      $transfer(s_j) = transfer(s_j) \cup \{(wp, seed)\}$   //store predicate relationships
      $seed = wp$
      if $seed$ is constant (i.e, $true$ or $false$) then exit inner for loop
    end for
  end if
end for
$marked = marked \cup \{s_1\}$
```

Fig. 2. Predicate localization during refinement.

3 Computing Abstractions

We describe the abstraction of the given C program by defining a transition system T. The transition system $T = (Q, I, R)$ consists of a set of states Q, a set of initial states $I \subseteq Q$, and a transition relation $R(q, q')$, which relates the current state $q \in Q$ to a next-state $q' \in Q$. The abstract model preserves the control flow in the original C program. Let $P = \{p_1, \ldots, p_k\}$ denote the union of the predicates active at various program locations ($\cup_s local(s)$). We first describe an abstraction scheme where each predicate p_i is assigned one unique Boolean variable b_i in the abstract model. Thus, the state space of the abstract model is $|L| \cdot 2^k$, where L is the set of control locations in the program. We call this scheme *abstraction without register sharing*. Next, we will describe a scheme where the number of Boolean variables needed to represent the predicates in P is equal to the maximum number of predicates active at any program location. Thus, the size of the abstract model is given by $|L| \cdot 2^{k'}$, where $k' = max_{1 \le i \le |L|} |local(s_i)|$. We call this scheme *abstraction with register sharing*. Due to the localization of predicates k' is usually much smaller than k, which enables faster model checking of the abstraction obtained using register sharing.

3.1 Abstraction Without Register Sharing

Let PC denote the vector of state variables used to encode the program counter. In abstraction without register sharing each predicate p_i has a state variable b_i in the abstract model. Each state in the abstraction corresponds to the valuation of $|PC| + k$ state variables, where k is the total number of predicates. In the initial state PC is equal to the value of the entry location in the original program. The state variables corresponding to the predicates are initially assigned non-deterministic Boolean values. Given a statement s_l and a predicate p_i the following cases are possible:

- s_l is either an assume statement or an assignment statement that does not assign to any variable in p_i. That is, after executing s_l the value of predicate p_i remains unchanged. Thus, in the abstract model the value of the state variable b_i remains unchanged after executing s_l. We denote the set of all statements where p_i is unchanged as $unc(p_i)$.

- s_l assigns to some variable in p_i. Let p_j denote the weakest pre-condition of p_i with respect to s_l. If the predicate p_j is active at s_l, that is $p_j \in local(s_l)$, and $(p_j, p_i) \in transfer(s_l)$, then after executing s_l, the value of predicate p_i is the same as the value of predicate p_j before executing s_l. In the abstract model this simply corresponds to transferring the value of b_j to b_i at s_l. If the predicate p_j is not active at s_l, then the abstract model assigns a non-deterministic Boolean value to b_i at s_l. This is necessary to ensure that the abstract model is an over-approximation of the original program.

We denote the set of all statements that can update the value of a predicate p_i as $update(p_i)$. The set of statements where the weakest pre-condition of p_i is available is denoted by $wpa(p_i)$. Using the localization information from Sec. 2.2, $wpa(p_i)$ is defined as follows: $wpa(p_i) := \{s_l | s_l \in update(p_i) \ \wedge \ \exists p_j.\ (p_j, p_i) \in transfer(s_l)\}$.

We use $inp(p_i)$ to denote the set of statements that assign a non-deterministic value v_i to the state variable b_i. This set is defined as $update(p_i) \backslash wpa(p_i)$. Let c_{il} denote the state variable corresponding to the weakest pre-condition of predicate p_i with respect to s_l. We use pc_l to denote that program counter is at s_l, that is $PC = l$. The next state function for the variable b_i is then defined as follows:

$$b_i' := \Big[\bigvee_{s_l \in unc(p_i)} (pc_l \wedge b_i) \Big] \vee \Big[\bigvee_{s_l \in wpa(p_i)} (pc_l \wedge c_{il}) \Big] \vee \Big[\bigvee_{s_l \in inp(p_i)} (pc_l \wedge v_i) \Big] \qquad (1)$$

Note that no calls to a decision procedure are needed when generating the next-state functions. All the required information is gathered during the refinement step itself by means of weakest pre-condition propagation.

Example: Consider the abstraction of the program in Fig. 3(a) with respect to the localization information given in Fig. 3(b). The predicate p_1 $(y \neq m+1)$ is updated at statement s_2, and its weakest pre-condition p_2 $(c \neq m)$ is active at s_2, and $(p_2, p_1) \in transfer(s_2)$. So the next state function for the state variable representing p_1 is given as follows: $b_1' := (pc_2 \wedge b_2) \vee ((pc_1 \vee pc_3 \vee pc_4) \wedge b_1)$. The other next state functions are given as follows: $b_2' := b_2$, $b_4' := b_4$, and $b_3' := (pc_1 \wedge b_4) \vee ((pc_2 \vee pc_3 \vee pc_4) \wedge b_3)$. The resulting abstraction is shown in Fig. 3 (c). For simplicity the control flow is shown explicitly in the abstraction.

Global constraint generation: The precision of the abstraction can be increased by finding the relationships between the predicates in $local(s)$ for some s. For example, in Fig. 3(b) the relationship between the predicates in $local(s_1)$ results in a *global constraint*, $b_2 \leftrightarrow \neg b_4$. This constraint holds in all states of the abstract model of Fig. 3 (c) as the Boolean variables b_2 and b_4 always represent the same predicate throughout the abstraction without register sharing. The abstraction without register sharing given in

```
s
1: x = c;
2: y = c + 1;
3: if (x == m)
4:  if (y != m+1)
5:   ERROR:;
```

(a)

local(s)	transfer(s)
$\{p_2, p_4\}$	$\{(p_2, p_2), (p_4, p_3)\}$
$\{p_2, p_3\}$	$\{(p_2, p_1), (p_3, p_3)\}$
$\{p_1, p_3\}$	$\{(p_1, p_1)\}$
$\{p_1\}$	

(b)

Abstraction
1: $b_3 = b_4$;
2: $b_1 = b_2$;
3: if (b_3)
4: if (b_1)
5: ERROR: ;

(c)

s	Mapping
1:	$\{p_2 : b_1, p_4 : b_2\}$
2:	$\{p_2 : b_1, p_3 : b_2\}$
3:	$\{p_1 : b_1, p_3 : b_2\}$
4:	$\{p_1 : b_1\}$
5:	

(d)

Abstraction
1: skip;
2: skip;
3: if (b_2)
4: if (b_1)
5: ERROR: ;

(e)

Global constraint for (c):
$b_2 \leftrightarrow \neg b_4$

Local constraint for (e):
$(PC = 1) \rightarrow (b_1 \leftrightarrow \neg b_2)$

(f)

Fig. 3. (a) C program. (b) Localization information for the program where p_1, p_2, p_3, p_4 denote the predicates $y \neq m+1, c \neq m, x = m, c = m$, respectively. (c) Abstraction with no register sharing. Boolean variable b_i represents the value of p_i in the abstraction. (d) Mapping of predicates in $local(s)$ for each s to the Boolean variables (register sharing). (e) Abstraction with register sharing. (f) Global constraint and Local constraint for abstractions in (c) and (e) , respectively.

Fig. 3(c) combined with the global constraint in Fig. 3(f) is sufficient to show that the `ERROR` label is not reachable in the C program given in Fig. 3(a). Note that we could have simplified the computation here by recognizing that $p_4 = \neg p_2$, which we omit for presentation purposes only.

The constraint generation is done only for some of the statements which are marked during the refinement (Fig. 2, line no. 17). We use SAT-based predicate abstraction [6, 17] to find the relationships between the predicates in $local(s)$ for such statements. This is the only time we use any decision procedure other than checking for the feasibility of traces. Due to the computational cost of enumerating the set of solutions, we only perform this computation for very small sets of predicates. Other relationships are then discovered on demand based on spurious abstract states [1].

3.2 Abstraction With Register Sharing

In abstraction with no register sharing, the state-space of the abstract model is $|L| \cdot 2^{|P|}$, where P is the set of predicates, and L is the set of locations in the given program. Thus, when the number of predicates is large, model checking of the abstraction can become a bottleneck even with a symbolic representation of the state space. We make use of the locality of predicates to speed up the model checking of the abstraction. This is done by reducing the number of (Boolean) state variables in the abstraction. The fact that each state variable in the abstract model is only locally useful can be used to represent different predicates in different parts of the program using the same state variable. We call the reuse of state variables in the abstract model *register sharing*.

Example: Consider the C program in Fig. 3(a) and the localization information in Fig. 3(b). The abstraction of this program with *no* register sharing in Fig. 3(c), contains four state variables, one for each predicate. However, the number of predicates active at any program statement is $max_{1 \leq i \leq 4} |local(s_i)| = 2$. Intuitively, it should be possible to create an abstraction with just two state variables.

The predicates p_2, p_4 are active at program location 1, so we introduce two Boolean variables b_1, b_2, to represent each of these predicates, respectively. After the execution of s_1, predicate p_4 is no longer active, and the state variable b_2 can be used to represent some other predicate. Predicate p_3 becomes active at s_2, so we can reuse the abstract variable b_2 to represent p_3 at s_2. In a similar fashion, b_1 can be reused to represent predicate p_1 at program locations s_3 and s_4. We use $p : b$ to denote that the predicate p is represented by the state variable b. The mapping of active predicates at each program location to the state variables is given in Fig 3(d).

The abstraction with register sharing is obtained by translating the predicate relationships in $transfer(s)$ for each s, according to the mapping discussed above. Continuing our example, $(p_4, p_3) \in transfer(s_1)$ in Fig. 3(b), the value of the state variable representing p_4 at s_1, must be transferred to the state variable representing p_3, afterwards. Since both p_4 and p_3 are represented by the same state variable b_2, the abstraction for s_1 does not alter the value of b_2. The abstraction using only two state variables (b_1, b_2) is shown in Fig 3(e). The `skip` statement means that the values of the state variables b_1 and b_2 remain unchanged for that statement.

Mapping predicates to state variables: Recall, that $p = \{p_1, \ldots, p_k\}$ denotes the set of predicates. Let $B = \{b_1, \ldots, b_l\}$ be the set of state variables in the abstraction, where l equals the maximum number of active predicates at any program location. For every statement s, the predicates relevant at s are mapped to unique state variables in B. Let map be a function that takes a statement s and a predicate p as arguments. If $p \in local(s)$, then the result of $map(s, p)$ is a state variable $b \in B$; otherwise, the result is $\perp$. Recall that $child(s)$ denotes the set of statements reachable from s in one step in the control flow graph. The constraints to be satisfied by map are as follows:

- Two distinct predicates which are active together at the same statement should not be assigned the same Boolean variable in the abstraction for that statement.

$$\forall s \forall p_i, p_j \in local(s)\ [p_i \neq p_j \rightarrow map(s, p_i) \neq map(s, p_j)]$$

- Consider statement s and $(p_1, p_2) \in transfer(s)$. By definition there exists $s' \in child(s)$ where p_2 is active, that is $p_2 \in local(s')$. This case is shown in Fig. 4(a). Suppose the predicate p_1 is mapped to b_i in s and p_2 is mapped to b_j in s'. The abstraction for the statement s will assign the value of b_i to b_j. So b_j should not be used to represent a predicate p_3, where $p_3 \neq p_2$, in any other successor of s. This is because there is no relationship between the value of the predicate p_1 at s and the predicate p_3 at s''. This constraint is shown in Fig. 4(b).

We now describe the algorithm which creates an abstraction in the presence of register sharing. Let $abs(s)$ be a set of Boolean pairs associated with each statement s. Intuitively, if $(b_l, b_m) \in abs(s)$, then in the abstraction the value of b_m after s terminates is equal to the value of b_l before the execution of s. Formally, $abs(s)$ is defined as follows:

$$abs(s) := \{(b_l, b_m) | \exists (p_i, p_j) \in transfer(s).\ b_l = map(s, p_i)\ \wedge$$

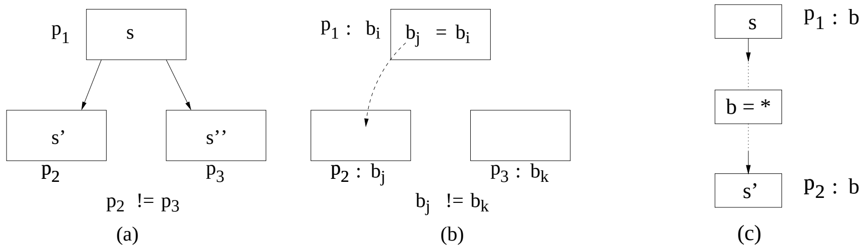


Fig. 4. (a) Statement s and two successors s' and s''. Predicates p_1, p_2, p_3 are active at s, s', and s'', respectively. (b) Abstraction with register sharing, where $(p_1, p_2) \in transfer(s)$. Predicate p_1, p_2 are mapped to b_i, b_j, respectively, in the abstraction. Predicate $p_3 \neq p_2$ should not be mapped to b_j for safe abstraction i.e., an over-approximation of original program. (c) Boolean variable b is used to represent two distinct predicates p_1 and p_2 on the same path. It is set to a * (non-deterministic value) between s and s' to ensure safe abstraction.

$$\exists s' \in child(s).\ b_m = map(s', p_j)\}.$$

Given a Boolean variable b_i and a statement s_l, the following cases are possible:

- s_l updates the value of b_i. That is, there exists a $b_j \in B$ such that $(b_j, b_i) \in abs(s_l)$. We denote the set of all statements which update b_i as $update(b_i)$. The function $rhs(s_l, b_i)$ returns the Boolean variable which is assigned to b_i in the statement s_l.

- s_l assigns a non-deterministic value to b_i. The set of all such statements is denoted by $nondet(b_i)$. In order to understand the use of this set, consider a Boolean variable b which is used to represent two distinct predicates p_1 and p_2 on the same path. Assume that b is not used to represent any other predicate between the statements s and s'. Since p_1 and p_2 are not related, the value of b when it is representing p_1 should not be used when b is representing p_2. So b is assigned a non-deterministic value between the path starting from s to s'. This is necessary to ensure that the abstraction is an over-approximation of the original program. This case is shown in Fig. 4(c).

- The value of b_i is a don't-care at statement s_l. The value of b_i is a don't care for all the statements which are not present in $update(b_i)$ or $nondet(b_i)$. In such cases, we set the value of b_i to false at these statements, in order to simplify its conjunction with the program counter variable to false. This simplifies the overall transition relation.

Given the above information the next state function for the variable b_i is defined as follows (we use an input v_i for introducing non-determinism and pc_l to denote $PC = l$):

$$b_i' := \Big[\bigvee_{s_l \in update(b_i)} (pc_l \wedge rhs(s_l, b_i))\Big] \vee \Big[\bigvee_{s_l \in nondet(p_i)} (pc_l \wedge v_i)\Big]. \quad (2)$$

Local constraint generation: The abstraction can be made more precise by relating the predicates in $local(s)$ for some s. For example, in Fig. 3(b) the predicates in $local(s_1)$ satisfy the constraint that $p_2 \leftrightarrow \neg p_4$. In order to add this constraint to the abstraction, we need to translate it in terms of the Boolean variables. The mapping given in Fig. 3(d) assigns Boolean variables b_1, b_2 to p_2, p_4, at s_1 respectively. This leads to a constraint $(PC = 1) \rightarrow (b_1 \leftrightarrow \neg b_2)$. This is called a *local constraint* as it is useful only when $PC = 1$. We cannot omit the $PC = 1$ term from the constraint as this would mean that $b_1 \leftrightarrow \neg b_2$ holds throughout the abstraction which is not correct due to register reuse.

The abstraction with register sharing in Fig. 3(e) combined with the local constraint in Fig. 3(f) is sufficient to show that the `ERROR` label is not reachable in the C program given in Fig. 3(a).

4 Dedicated State Variables

Register sharing enables the creation of abstract models with as few Boolean variables as possible which enables more efficient model checking of the abstractions. However, register sharing might also result in a large number of refinement iterations as described in the following. Consider a sequence SE of statements from s to s', which does not modify the value of a predicate p. Suppose p is localized at the statements s, s', but not at any intermediate statement in SE. In abstraction with register sharing, it is possible that p is represented by two different Boolean variables b_1 and b_2 at s and s', respectively. Because the value of p remains unchanged along SE, the value of b_1 at s should be equal to the value of b_2 at s'. If this is not tracked however, we may obtain a spurious counterexample by assigning different values to b_1 at s and b_2 at s'. This leads to a refinement step, which localizes the predicate p at every statement in SE, to ensure that the value of predicate p does not change along SE in subsequent iterations. We should note that such behavior is handled in the abstraction *without* register sharing approach through the use of the *unchanged* set denoted by *unc* in Eqn. (1) described earlier.

If p is discovered frequently in different parts of the program through various spurious counterexamples, then using the abstraction with register sharing will lead to many abstraction refinement iterations. This problem can be avoided, if p is represented by exactly one Boolean variable b in a large scope of the abstraction. This is because the value of b will not be changed by any statement in SE, and thus, the value of b at s' will be the same as that at s. We call a Boolean variable which represents only one predicate throughout the abstraction a *dedicated state variable*. The next state function for a dedicated state variable b is computed using Eqn. (1).

Hybrid approach: Initially, when a predicate is discovered it is assigned a Boolean variable, which can be reused for representing different predicates in other parts of the abstraction. If the same predicate is discovered through multiple counterexamples in the various parts of the program, then it is assigned a dedicated Boolean variable for a global or functional scope of the program depending on the variables used in the predicate. The decision about when to assign a dedicated Boolean variable to a predicate is done by making use of the following heuristic.

For each predicate p, let $usage(p,i)$ denote the number of statements where p is localized in the iteration number i of the abstraction refinement loop. If $usage(p,i)$ exceeds a certain user-defined threshold TH, then p is assigned a dedicated Boolean variable. If $TH = 0$, then every predicate will be assigned a dedicated state variable as soon as it is discovered. This is similar to performing abstraction with no register sharing for all state variables. On the other hand, if $TH = |L| + 1$, where $|L|$ is the total number of statements in the program, then none of the predicates will be assigned a dedicated state variable. This allows complete reuse of the abstract variables, which is similar to abstraction with register sharing. For any intermediate value of TH we have a *hybrid* of abstraction with and without register sharing.

In the hybrid approach, it is possible to have global constraints on the dedicated state variables. This saves refinement iterations where the same constraint is added locally in various parts by means of counterexamples. We can still have local constraints on the state variables which are reused. Furthermore, we hope to discover as early as possible whether a predicate should be given a dedicated state variable by having a low threshold for the early iterations of the abstraction refinement loop, which increases as the number of iterations increases. Predicting early on that a predicate may need a dedicated state variable reduces the number of abstraction refinement iterations substantially.

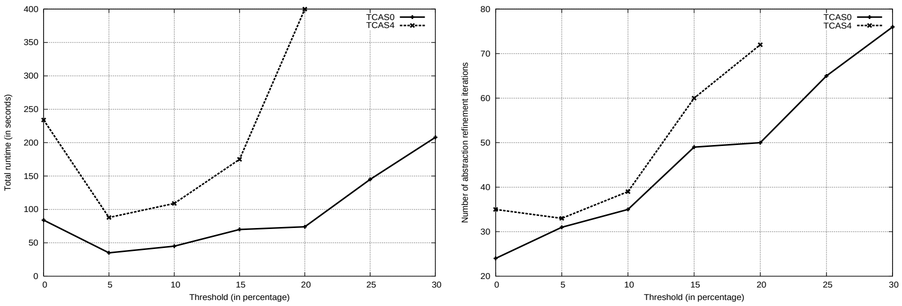
5 Experimental Results

We have implemented these techniques in NEC's F-SOFT [15] verification tool. All experiments were performed on a 2.8*GHz* dual-processor Linux machine with 4GB of memory. We report our experimental results on the TCAS and Alias case studies. TCAS (Traffic Alert and Collision Avoidance System) is an aircraft conflict detection and resolution system used by all US commercial aircrafts. We used an ANSI-C version of a TCAS component available from Georgia Tech. Even though the pre-processed program has only 1652 lines of code, the number of predicates needed to verify the properties is non-trivial for both F-Soft and BLAST. We checked 10 different safety properties of the TCAS system. Alias is an artificial benchmark which makes extensive use of pointers. Each property was encoded as a certain error label in the code. If the label is not reachable, then the property is said to hold. Otherwise, we report the length of the counterexample in the ”Bug” column in Table 1. All CPU times are given in seconds. For all runs, we set a time limit of one hour for the analysis.

5.1 Predicate Localization, Register Sharing, and Dedicated State Variables

We first experimented with no localization of predicates. However, this approach did not scale, as the abstraction computation becomes a bottleneck. We next experimented with localization of predicates using weakest pre-conditions. The results of applying only localization and abstraction without register sharing is shown under the ”Localize” heading in the Table 1. The ”Time Abs MC” column gives the total time, followed by the breakup of total time into the time taken by abstraction (Abs), model checking (MC), respectively. We omit the time taken by refinement, which is equal to Time - (Abs + MC) for each row. The ”P” and the ”I” columns give the total number of predicates, and the total number of iterations, respectively. Two observations can be made from the ”Localize” results: 1) Due to the localization of predicates, the abstraction computation is no longer a bottleneck. 2) Model checking takes most of the time, since for each predicate a state variable is created in the abstract model. Note that the model checking step is the cause of the timeouts in three rows under the ”Localize” results.

Next, we experimented with register sharing. The number of state variables in the abstraction was reduced, and the individual model-checking steps became faster. However, as discussed in Sec. 4 this approach resulted in too many abstraction refinement iterations. This problem was solved by discovering on-the-fly whether a predicate should be assigned a dedicated state variable, that is, a state variable which will not be reused.



(a) (b)

Fig. 5. (a) Variation in the total runtime with the threshold. (b) Variation in the total number of abstraction refinement iterations with the threshold.

A dedicated state variable is introduced for a predicate whose usage exceeds a progressively increasing threshold, starting at 5% of the total number of program locations.

The results of combining these multiple techniques is given under the ”Combined” heading in Table 1. The meaning of ”Time Abs MC” and the ”I” column is the same as before. The ”P Max Ded” column gives the total number of predicates (P), followed by the maximum number of predicates active at any program location (Max), and the total number of state variables which represent exactly one predicate, that is, dedicated state variables (Ded). Observe that the time spent during model checking (MC) has reduced significantly as compared to the ”Localize” column.

Effect of varying the threshold: We also experimented with the TH (threshold) parameter, which is used to determine when a predicate is assigned a dedicated state variable. Fig. 5(a) shows the variation of the total runtime with the initial value for the threshold. When the threshold is equal to zero every predicate is assigned a dedicated state variable. This results in too many state variables in the abstract model causing the total runtime to be high. However, as the threshold is increased, the number of abstraction refinement iterations starts to increase as shown in Fig. 5(b). The best runtime in our experiments has so far been obtained for an initial threshold of 5%. Even such a small value for the threshold is effective in separating the predicates which are *globally* relevant from those which are *locally* useful. As the threshold is further increased very few predicates are assigned dedicated state variables. One of the main advantages of choosing a small initial threshold is that we are able to decide early on whether a predicate may need a dedicated state variable. If we start with a higher initial threshold, the number of additional iterations needed for a single predicate to receive a dedicated state variable increases too much.

Variable reuse ratio: In our implementation the map function (Section 3.2) is computed incrementally, as new predicates are discovered. Suppose during refinement a predicate p gets added to $local(s)$ for some s. In order to find a state variable to represent the value of p at s, we first check if some existing state variable can be reused without violating the constraints described in Section 3.2. Let the total number of times reuse is possible

Bench	Localize					Combined							BLAST					Bug
-mark	Time	Abs	MC	P	I	Time	Abs	MC	P	Max	Ded	I	Time	P	Max	Avg	I	
TCAS0	245	7	196	71	32	**36**	5	15	65	26	18	31	96	85	24	10	33	-
TCAS1	1187	15	1069	108	44	**161**	9	118	96	35	25	38	256	137	43	17	42	-
TCAS2	952	10	882	74	38	**104**	25	51	95	31	24	36	148	108	31	11	40	-
TCAS3	940	15	864	91	36	**46**	17	17	73	22	15	33	172	101	26	10	44	152
TCAS4	1231	13	1111	97	39	**88**	9	48	90	34	25	32	182	149	38	13	51	166
TCAS5	1222	11	1128	79	41	141	8	98	98	37	29	31	**105**	114	31	10	33	-
TCAS6	TO	20	2270	117	49	330	16	266	109	40	33	40	**293**	158	41	14	69	179
TCAS7	1758	16	1627	79	47	**64**	10	29	94	28	21	33	287	125	30	11	63	160
TCAS8	TO	21	1988	84	51	**119**	13	68	106	34	27	41	181	116	31	11	46	-
TCAS9	TO	26	3349	113	58	**250**	14	186	106	34	27	44	322	140	40	14	61	179
ALIAS	50	6	33	61	11	**6**	2	1	55	25	15	9	·	·	·	·	·	-

Table 1. Results for: 1) Localization, abstraction without register sharing (”Localize”) . 2) Localization, abstraction with register sharing, dedicated state variables (”Combined”). 3) BLAST with interpolation (”BLAST”). A ”-” indicates that the property holds. A ”·” indicates that the benchmark could not be handled properly. A ”TO” indicates a timeout of 1hr. We report the statistics observed before timeout occurs.

be R. If no existing state variable can be used, we introduce a new state variable for representing the value of p at s. Let the total number of times a new state variable is introduced be C. The ratio $R/(C+R)$ measures the effectiveness of variable reuse in controlling the total number of state variables. The value of this ratio is 88% on average across the `TCAS` benchmarks and 81% for the `ALIAS` benchmark.

5.2 Comparison with BLAST

We first ran BLAST in the default mode without any options. However, the default predicate discovery scheme in BLAST fails to find the new set of predicates during refinement, and terminates without (dis)proving any of the TCAS properties. Next, we tried the Craig interpolation [14] options (`craig1` and `craig2`) provided by BLAST. The BLAST manual recommends the use of `predH7` heuristic with Craig interpolation. Of the various options to BLAST, `craig2` and `predH7` result in the best performance when checking the TCAS properties. Table 1 gives the result of running BLAST with these options under the ”BLAST” heading. The ”P Max Avg” column gives the total number of predicates (P), followed by the maximum (Max) and the average (Avg) number of predicates active at any program location (rounded to the nearest integer).

The best runtimes are shown in bold in Table 1. Note that the ”Combined” technique of F-Soft outperforms BLAST on 9 out of 11 benchmarks, and the number of iterations required by ”Combined” is less than that for ”Blast” in all cases. Recall that the size of the abstraction is exponential in the maximum number of active predicates (Max). This number is comparable for both BLAST and F-Soft, even though BLAST makes use of a more complex refinement technique based on the computation of interpolants.

6 Conclusions and Future Work

The application of the predicate abstraction paradigm to large software depends crucially on the choice and usage of the predicates. If all predicates are tracked globally in the program, the analysis often becomes intractable due to the large number of predicate relationships. In this paper we described various techniques for improving the overall performance of the abstraction refinement loop. We presented experimental results in our F-SOFT [15] toolkit using the techniques of predicate localization, register sharing and dedicated state variables, and showed how a combination of these techniques allowed us to check properties requiring a large number of predicates.

There are a number of interesting avenues for future research. Theoretical comparison between the use of interpolants [14] and the use of weakest pre-conditions for localization of predicates will be useful. Other techniques for finding the right balance between the predicates whose values are tracked locally and the predicates whose values are tracked globally are worth further investigation.

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

References

1. T. Ball, B. Cook, S. Das, and S. Rajamani. Refining approximations in software predicate abstraction. In *TACAS 04*, pages 388–403. Springer, 2004.
2. T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation*, pages 203–213, 2001.
3. T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS 01*, volume 2031, 2001.
4. T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*. Springer, 2001.
5. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE 03*, pages 385–395. IEEE, 2003.
6. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25:105–127, Sep–Nov 2004.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
8. William Craig. Linear reasoning. In *Journal of Symbolic Logic*, pages 22:250–268, 1957.
9. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification*, LNCS 1633, pages 160–171. Springer, 1999.
10. E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
11. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV 97*, pages 72–83. Springer, 1997.
12. A. Gupta, M.K. Ganai, P. Ashar, and Z. Yang. Iterative abstraction using SAT-based BMC with proof analysis. In *International Conference on Computer Aided Design (ICCAD)*, 2003.
13. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02*, pages 58–70, 2002.
14. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*, pages 232–244. ACM Press, 2004.
15. F. Ivančić, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. In *Symposium on Leveraging Applications of Formal Methods*, 2004.
16. H. Jain, D. Kroening, and E. Clarke. Verification of SpecC using predicate abstraction. In *MEMOCODE 04*, pages 7–16. IEEE, 2004.

17. S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *CAV 03*, pages 141–153. Springer, 2003.
18. Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV 00*, number 1855 in LNCS, 2000.
19. R. Rugina and M.C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI 00*, pages 182–195, 2000.
20. Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *TACAS 99*, pages 178–192, 1999.
21. A. Zaks, F. Ivančić, H. Cadambi, I. Shlyakhter, Z. Yang, M. Ganai A. Gupta, and P. Ashar. Range analysis for software verification. *Submitted for publication*, 2004.