Polymorphic Predicate Abstraction

THOMAS BALL
Microsoft Research
TODD MILLSTEIN
University of California, Los Angeles
and
SRIRAM K. RAJAMANI
Microsoft Research

Predicate abstraction is a technique for creating abstract models of software that are amenable to model checking algorithms. We show how polymorphism, a well-known concept in programming languages and program analysis, can be incorporated in a predicate abstraction algorithm for C programs. The use of polymorphism in predicates, via the introduction of symbolic names for values, allows us to model the effect of a procedure independent of its calling contexts. Therefore, we can safely and precisely abstract a procedure once and then reuse this abstraction across multiple calls and multiple applications containing the procedure. Polymorphism also enables us to handle programs that need to be analyzed in an open environment, for all possible callers. We have proved that our algorithm is sound and have implemented it in the C2BP tool as part of the SLAM software model checking toolkit.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking, reliability, validation*

General Terms: Algorithms, Reliability, Verification

Additional Key Words and Phrases: Software model checking, predicate abstraction, polymorphism

1. INTRODUCTION

Predicate abstraction [Graf and Saïdi 1997; Das et al. 1999] is a technique for automatically creating a finite-state system (for which a fixpoint analysis will terminate) from an infinite-state (or very large finite-state) system (for which a fixpoint analysis will, in general, not terminate). With predicate abstraction,

The work of T. Millstein was performed while he was at Microsoft Research and the University of Washington.
Authors' addresses: T. Ball and S. K. Rajamani, Microsoft, One Microsoft Way, Redmond, WA 98052; email: {tball,sriram}@microsoft.com; T. Millstein, UCLA Computer Science Department, 4531D Boelter Hall, Los Angeles, CA 90095-1596; email: todd@cs.ucla.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2005 ACM 0164-0925/05/0300-0314 $5.00

the concrete states of a system are mapped to abstract states according to their valuation under a finite set of predicates.

Predicate abstraction has been used to construct abstractions of hardware and protocol designs in the model-checking community. The technique has been recently applied to software written in general-purpose programming languages [Visser et al. 2000; Ball et al. 2001a; Henzinger et al. 2002; Flanagan and Qadeer 2002]. In previous work, we introduced an automatic predicate abstraction algorithm for C programs, as implemented in the C2BP tool [Ball et al. 2001a]. Given a C program P and a set $E = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ of pure Boolean C expressions over the variables in P, C2BP automatically constructs a *Boolean program* abstraction $BP(P, E)$, a program that has identical control structure to P (including procedures and procedure calls) but contains only Boolean variables. The program $BP(P, E)$ contains n Boolean variables (*b-variables*) $V = \{b_1, b_2, \ldots, b_n\}$, where each b-variable b_i represents the value of predicate φ_i. Each b-variable in V has a *three-valued* domain: **false**, **true**, and $*$(representing "don't know"). Boolean programs are amenable to model-checking algorithms, including the one implemented in the BEBOP tool [Ball and Rajamani 2000]. The combination of the C2BP and BEBOP tools can be used to discover inductive invariants in a C program that are boolean functions over the predicates in E.

The C2BP algorithm presented in our previous work is unique in that it supports *modular* abstraction of procedures and procedure calls. Each procedure R is abstracted by C2BP into an abstract version R', without requiring access to its callers. Each call to R can then separately be abstracted to a call to R', without requiring access to the implementations of either R or R'. This approach contrasts with other work on predicate abstraction in the presence of procedures, in which procedures are simply inline-expanded.

Modular abstraction of procedures provides a number of important benefits. First, it allows abstraction to be robust to program evolution. For example, the abstraction of some procedure R need not be modified as a program evolves to include new calls to the procedure; R's original abstraction can be safely reused by the new callers. Second, modular abstraction supports independent code development. For example, the implementer of R can produce its abstraction once, in isolation from all client programs that use R. Given R's abstraction, those client programs can be safely abstracted without ever requiring access to R's source code. Third, modular abstraction does not incur the potential for code explosion that is possible with procedure inlining. Finally, modular abstraction of procedures naturally supports (mutually) recursive procedures.

While our earlier C2BP algorithm allows a procedure to be abstracted in isolation from its callers, some knowledge of these callers is nonetheless necessary in practice, in order for the procedure's abstraction to be sufficient to prove the desired properties about these callers. As an example, consider the following procedure, which is the identity function on integers:

```
int id(int x) {return x;}.
```

Suppose we would like the Boolean program abstraction to prove that the value returned by `id` is 5 when some client `f1` invokes it. In this case, we must add the

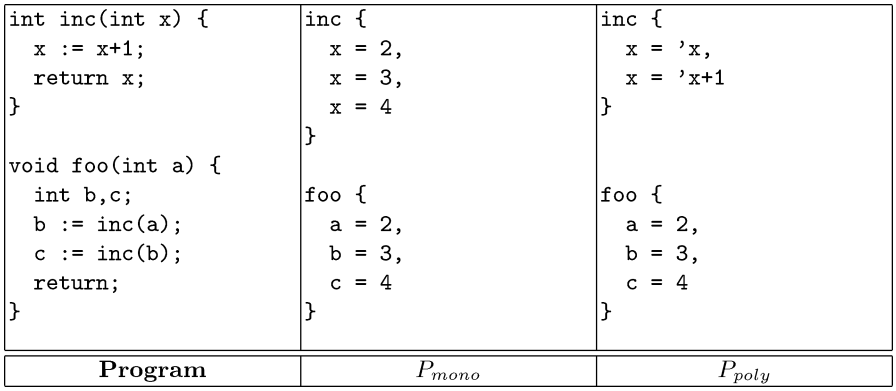
predicate $(x = 5)$ to E. Similarly, to prove that the value returned by `id` is 73 when another client `f2` invokes it, we must add the predicate $(x = 73)$ to E. In general, a separate predicate is required for each concrete value to be tracked.

This dependence of a procedure's abstraction on its callers limits the benefits of a modular abstraction process described above. A procedure will likely need to be re-abstracted (to incorporate new predicates) whenever a call to the procedure is added. This makes it difficult to reuse a procedure's abstraction as a program evolves or in multiple programs containing the procedure. In the limit, the number of predicates required in the abstraction of a procedure will be proportional to the number of call sites of the procedure. Because the abstraction process is exponential in the number of predicates, this requirement can become prohibitively expensive.

The above example also illustrates an expressiveness limitation of the predicate language in our earlier work. Using only pure Boolean C expressions, the model checker will not be able to deduce a fact such as "`id` returns the same value that it is passed." Instead, the model checker can deduce only *instantiations* of this fact, for specific actual parameter values. Deducing facts about all possible callers is especially critical when model checking programs in an open environment. For example, the SLAM software model checking toolkit [Ball and Rajamani 2002b], which employs the C2BP tool, analyzes device drivers without also analyzing the entire Windows operating system in which the drivers run. To prove properties of device drivers conservatively in such a setting, SLAM must analyze their source code under the assumption that the drivers' routines can be called from an arbitrary context.

In this article, we show how to resolve all of these previous limitations of predicate abstraction by incorporating a form of polymorphism. We extend the predicate language to allow the use of *symbolic constants*, which are names given to the initial values of a procedure's formal parameters. A predicate containing a symbolic constant is said to be *polymorphic*. In our example above, we can use the single polymorphic predicate $(x = {}'x)$ (where $'x$ is the symbolic constant for x) to capture the relevant information about `id` in the resulting Boolean program. The predicate abstraction algorithm is extended to appropriately handle polymorphic predicates, substituting concrete values for symbolic constants during the abstraction of each procedure call (just as types are substituted for type variables at each call site of a type-polymorphic function, in languages supporting parametric polymorphism). In this way, all calls to `id` can reuse its single polymorphic abstraction. In the presence of pointers, we also support symbolic constants that refer to dereferences of formal parameters, in order to talk about the initial values of locations *reachable* from a procedure's formal parameters.

The generalized predicate abstraction algorithm is implemented in our C2BP tool, which is part of the SLAM toolkit [Ball and Rajamani 2002b]. The toolkit automatically checks whether a C program satisfies a given set of temporal safety properties [Ball and Rajamani 2001]. SLAM involves an iterative process that employs three components: C2BP, the BEBOP model checker, and a predicate discoverer. This article focuses on the polymorphic predicate abstraction algorithm employed by C2BP. While beyond the scope of this article, the other



```
int inc(int x) {
  x := x+1;
  return x;
}

void foo(int a) {
  int b,c;
  b := inc(a);
  c := inc(b);
  return;
}
```

```
inc {
  x = 2,
  x = 3,
  x = 4
}

foo {
  a = 2,
  b = 3,
  c = 4
}
```

```
inc {
  x = 'x,
  x = 'x+1
}

foo {
  a = 2,
  b = 3,
  c = 4
}
```

Program	P_{mono}	P_{poly}

Fig. 1. A simple program and two sets of predicates, P_{mono} and P_{poly}.

two components naturally support polymorphic predicates as well. The model checker BEBOP need not be modified at all: as far as BEBOP is concerned, it is immaterial that the predicates are polymorphic, since BEBOP does not interpret what the *b*-variables in a Boolean program represent. The issue of how SLAM discovers predicates for use by C2BP is an interesting one, and it is discussed in detail elsewhere [Ball and Rajamani 2002a]. SLAM's predicate discovery algorithm can naturally generate polymorphic predicates as required by polymorphic predicate abstraction.

Section 2 informally reviews by example how C2BP performs *monomorphic* predicate abstraction for C programs and then presents the same example using polymorphic predicate abstraction. Section 3 introduces a core language used to formally explain the predicate abstraction algorithm, and Appendix A formally describes the three-valued logic used in this language. Section 4 presents the technical details of our algorithm, in the case when programs do not involve pointers, and Section 5 extends the algorithm to accommodate pointers. Section 6 discusses related work and Section 7 concludes the article. Appendix B proves that the modular C2BP abstraction algorithm is sound.

2. EXAMPLE

Figure 1 shows a simple example of a C program in which procedure `foo` calls an increment procedure `inc` twice.[1] Our goal is to prove that if `foo` is called with the value 2, then the value of `c` in `foo` will be 4. Consider the set of predicates P_{mono} in the figure. The predicates are partitioned between the two procedures. Procedure `inc` will be abstracted with respect to the first three predicates, and `foo` will be abstracted with respect to the last three. It is also possible to provide a set of *global* predicates to be employed during the abstraction of all procedures in a program, but this example contains none.

Figure 2 shows the Boolean program that C2BP produces when given the C program and the predicate set P_{mono} from Figure 1. The Boolean program has the same control-flow structure as the C program and contains a *b*-variable for

[1]Throughout, we use ":=" for the assignment operator and "=" for the equality operator.

```
bool,bool,bool inc(bool {x=2}, bool {x=3},
                   bool {x=4}) {
  {x=2}, {x=3}, {x=4}
      := choose(false, {x=2}|{x=3}|{x=4}),
         choose({x=2}, !{x=2}),
         choose({x=3}, !{x=3});
  return {x=2},{x=3},{x=4};
}

void foo(bool {a=2}) {
  bool {b=3},{c=4};
  bool prm1, prm2, prm3;
  bool ret1, ret2, ret3;
  ... // continued in the next column
```

```
  prm1 := choose({a=2}, !{a=2});
  prm2 := choose(false, {a=2});
  prm3 := choose(false, {a=2})
  ret1, ret2, ret3 := inc(prm1, prm2, prm3);
  {b=3}  := choose(ret2, !ret2);

  prm1 := choose(false, {b=3});
  prm2 := choose({b=3}, !{b=3});
  prm3 := choose(false, {b=3});
  ret1, ret2, ret3 := inc(prm1, prm2, prm3);
  {c=4}  := choose(ret3, !ret3);
  return;
}
```

Fig. 2. Boolean program abstraction created by the C2BP tool, given the program shown in Figure 1 and predicates P_{mono}.

each predicate input to C2BP.[2] A *b*-variable whose associated predicate refers only to formal parameters and global variables of the original program is a formal parameter, called a *b-parameter*, in the Boolean program. For example, the procedure `inc` takes three formal *b*-parameters, `{x=2}`, `{x=3}`, and `{x=4}`, corresponding to the three predicates in P_{mono} that mention the formal parameter x. The procedure returns three *b*-variables that represent the updated values of the predicates about x. Procedure `foo` has a formal *b*-parameter`{a=2}`. It also has local *b*-variables for the predicates in P_{mono} involving the local variables b and c.

C2BP creates this Boolean program by separately translating each statement s of the C program into one or more statements S in the Boolean program. The statements in S conservatively model the effect of s on each predicate that is in scope at the corresponding point in the C program. Consider the assignment statement `x := x+1` in the `inc` procedure. C2BP discovers that the predicate $(x = 3)$ will be true (false) after the assignment if the predicate $(x = 2)$ is true (false) before the assignment. This results in the translation (as part of the parallel assignment in `inc`): `{x=3} := choose({x=2}, !{x=2})`. The `choose` function is included in every Boolean program and is defined as follows:

```
bool choose(bool pos, bool neg) {
     if (pos) return true;
     else if (neg) return false;
     else return *;
}.
```

The `pos` *b*-parameter of the `choose` function represents sufficient conditions for the truth of a predicate, while `neg` represents sufficient conditions for the falsehood of a predicate. C2BP guarantees that `choose` is never called with both *b*-parameters evaluating to true. Both *b*-parameters may evaluate to false because the predicates being modeled are not strong enough to provide a definite

[2]Boolean programs permit an identifier to be of the form $\{p\}$, where p is an arbitrary string. This is useful for naming *b*-variables with the exact predicates they represent.

answer, or because the theorem proving machinery that C2BP uses is incomplete. In that case, the `choose` function conservatively returns $*$, representing the "don't know" value.

The abstraction process is modular. For example, the `inc` procedure is abstracted without knowledge of callers like `foo`. Each call is abstracted separately, given only `inc`'s abstraction. For example, consider the first procedure call `b := inc(a)` in procedure `foo`. In the C program, the actual parameter passed to `inc` is the expression a, whose value is assigned to the formal parameter x of `inc`. By substituting the actual parameter a for formal x in the predicate $(x = 2)$, C2BP determines that if the predicate $(a = 2)$ is true (false) before the call then the predicate $(x = 2)$ is true (false) at the entry of `inc`, resulting in the statement `prm1 := choose({a=2}, !{a=2})`. The b-variable `prm1` is then sent as the actual parameter for the first b-parameter of `inc` in the Boolean program, which represents the predicate $(x = 2)$. The other b-parameter values are determined similarly via assignments to the b-variables `prm2` and `prm3`. There are three return predicates in `inc`: $(x = 2)$, $(x = 3)$ and $(x = 4)$. Because the result of the call is assigned to b, C2BP determines that the predicate $(b = 3)$ will be true (false) after the call if $(x = 3)$ (represented by `ret2` in `foo`) is true (false) at that point. Thus, C2BP generates the assignment `{b=3} := choose(ret2, !ret2)`.

The Boolean program in Figure 2 was created in a completely modular way. In addition, it achieves our goal: when BEBOP model checks that Boolean program, it will determine that if `foo` is passed the value 2, then the variable `c` will have the value 4. Unfortunately, deducing this property of `foo` has required us to abstract `inc` in a way that is not of general use. The predicates $(x = 2)$ and $(x = 3)$ are necessary so BEBOP can determine that the return value is 3 when the value 2 is passed into `inc`. Similarly, the two predicates $(x = 3)$ and $(x = 4)$ are necessary so BEBOP can determine that the return value is 4 when the value 3 is passed into `inc`. An entirely different set of predicates would be required if we later wanted to reason about the value of `c` when `foo` is passed the value 5, thereby forcing us to re-abstract `inc`. Similarly, if the program evolves to contain other calls to `inc`, then `inc` will have to be re-abstracted with yet more predicates. Finally, there is no way to determine the effect of `inc` independent of a particular set of calling contexts, which is critical when attempting to reason about partial programs containing `inc`.

2.1 Polymorphic Predicate Abstraction

Although calls may differ in the values of arguments passed to a procedure, they often rely on the same underlying abstract *transfer function* of the procedure. The idea of polymorphic predicate abstraction is to employ predicates that allow the Boolean program abstraction to directly model this transfer function, which can then be shared across all call sites. In the example of the `inc` procedure, we wish to know that the return value of `inc` is always one more than the initial value of the formal parameter x of `inc`. Symbolic constants offer a way to do this. We denote the value of x at entry to `inc` by the symbolic constant $'x$. To precisely summarize the effect of `inc` for all possible call sites then requires only two predicates: $x = 'x$ and $x = 'x + 1$. The predicate set P_{poly} in Figure 1

```
bool,bool inc() {
  bool {x='x}, {x='x+1};
  {x='x} := true;
  {x='x}, {x='x+1} :=
    choose(false, {x='x}|{x='x+1}),
    choose({x='x}, !{x='x} );
  return {x='x}, {x='x+1};
}
```

```
void foo(bool {a=2}) {
  bool {b=3},{c=4};
  bool ret1, ret2;
  ret1, ret2 := inc();
  {b=3} := choose({a=2}&ret2,
                  ({a=2}&!ret2)|(!{a=2}&ret2));
  ret1, ret2 := inc();
  {c=4} := choose({b=3}&ret2,
                  ({b=3}&!ret2)|(!{b=3}&ret2));
  return;
}
```

Fig. 3. Boolean program abstraction created by the C2BP tool, given the input C program shown in Figure 1 and predicates P_{poly}.

shows such polymorphic predicates. At each call site, C2BP will bind $'x$ to the appropriate actual parameter value to get the desired effect.[3]

Figure 3 shows the Boolean program created by C2BP when applied to the C program and the predicate set P_{poly}. This Boolean program is similar to its monomorphic counterpart, but the use of symbolic constants allows it to prove much stronger properties about the original `inc` procedure. In particular, the Boolean program is able to model the fact that `inc` returns a value that is one greater than the original value of x, represented by $'x$. For example, consider the abstraction of the statement `x:=x+1`. The Boolean program records the fact that $(x = 'x + 1)$ is true (false) after that statement if $(x = 'x)$ is true (false) beforehand.

As before, the abstraction process is completely modular. Predicates involving symbolic constants $'x$ get interpreted appropriately at each call site, allowing callers to retain precision. For example, at the point of the `b := inc(a)` call in `foo`, the returned predicate $(x = 'x+1)$ (represented by `ret2`) gets interpreted as $(b = a + 1)$, since the returned variable x is assigned to b at the call site and the original value of x is equivalent to the value of the actual parameter a. This interpretation allows `foo`'s abstraction to learn that b has the value 3 after the call if a has the value 2 beforehand.

The abstractions of `inc` and `foo` in Figure 3 achieve our goal of showing that if `foo` is passed the value 2 then `c` has the value 4. However, the abstraction of `inc` can as easily be used to reason about other calling contexts. For example, modifying `foo`'s abstraction to reason about a call to `foo(5)` does not require reabstraction of `inc`. Similarly, as the program evolves to incorporate more calls to `inc`, these calls can safely and precisely use the abstraction of `inc` in Figure 3.

3. CORE LANGUAGE

To simplify the presentation of the central ideas in our approach, we focus our attention on a small core language containing procedures and references (but without type casts, pointer arithmetic, structures, arrays, unions, and explicit allocation and deallocation). Figure 4 presents the syntax of the core language.

[3]It is also possible to perform a polymorphic abstraction of `foo`, with the predicates $(a = 'a)$, $(b = 'a + 1)$, and $(c = 'a + 2)$. We retain `foo`'s monomorphic predicates for simplicity.

Types:	τ	::=	**void** | **bool** | **int** | **ref** τ
Expressions:	e	::=	$c \mid x \mid e_1\ op\ e_2 \mid \&x \mid *\cdots *x$
Declaration:	d	::=	$\tau\ \ x_1, \cdots, x_n;$
Statements:	s	::=	**skip** | **goto** L | L: s
		|	**branch** $\overline{s_1} \,\|\|\, \cdots \,\|\|\, \overline{s_n}$ **end**
		|	**assume**(e)
		|	**return** $x_1, \cdots, x_n$
		|	$*x\ :=\ e$
		|	$x_1, \cdots x_n\ :=\ e_1, \cdots, e_n$
		|	$x_1, \cdots, x_m := id(e_1, \cdots, e_n)$
Statement Sequences:	$\overline{s}$	::=	$s_1; \cdots s_n;$
Procedure:	p	::=	$\tau'_1, \cdots, \tau'_m\ id\ (f_1 : \tau_1, \cdots, f_n : \tau_n)$ **{** $d_1 \cdots d_q\ \overline{s}$ **}**
Program:	g	::=	$d_1 \cdots d_m\ p_1 \cdots p_n$

Fig. 4. A core language containing references and procedures.

The core language contains void, Boolean, integer, and reference types. The Booleans are *three-valued*, and we use Kleene's three-valued logic to interpret them (see Appendix A for a formal description). The expressions include Boolean and integer constants (ranged over by metavariable c), variables (ranged over by metavariable x), and the usual arithmetic and binary operations (ranging over metavariable op). We also provide two pointer expressions, the ability to create a reference to a variable and the ability to dereference a pointer variable as many times as its type allows. All variables must be declared before being used. Boolean variables are assumed to initially hold the value $*$; other variables are assumed to initially hold an arbitrary value of the appropriate type.

The statements include the **skip** statement, **goto** statement, and labeled statements. The **branch** statement nondeterministically selects one of its n branches to execute. Having a nondeterministic conditional statement in the language of Boolean programs (which, as discussed in Section 4, is a subset of this core language) allows us to easily model incomplete information about the original program's control flow. The **assume** statement is the dual of the `assert` statement found in some languages: the **assume** statement silently terminates execution if its expression evaluates to **false**. The **branch** and **assume** statements can be used to implement the common `if-then-else` and C-style `switch` statements. For example, the statement "`if` e `then` $\overline{s_1}$ `else` $\overline{s_2}$" is implemented as "**branch assume**(e); $\overline{s_1}$; ‖ **assume**$(!e)$; $\overline{s_2}$; **end**".

There are three forms of assignment statement in the language. The $(*x := e)$ form is an indirect assignment through a variable with reference type. For ease of exposition, we do not allow more than one dereference on the left-hand side of the assignment. This does not limit expressiveness. The second form is a parallel assignment of n expressions to n variables. Finally, the language contains a procedure call statement, in which the procedure can return a tuple of m values.

4. POLYMORPHIC PREDICATE ABSTRACTION ALGORITHM

This section presents our polymorphic predicate abstraction algorithm for the core language without reference types and associated expressions and

statements. We are given a program P in the core language and a set $E = \{\varphi_1, \ldots, \varphi_n\}$ of pure Boolean expressions over the variables in P. Each formal parameter has an associated symbolic constant, which may also be referred to in the predicates in E. The goal of the algorithm is to create a Boolean program abstraction $BP(P, E)$ that conservatively (yet as precisely as possible) represents the effect of each statement in P on each predicate in E that is in scope at that statement. The language of Boolean programs is simply the language of Figure 4, but with integers, reference types, and all associated expressions and statements removed.[4]

After defining some preliminaries, we discuss the concepts of weakest preconditions and predicate strengthening that form the basis of our abstraction algorithm. We then present the syntax-directed abstraction of the basic program statements, which follows our earlier algorithm for C2BP [Ball et al. 2001a]. Finally, we present the abstraction of procedure calls, which extends our earlier algorithm to safely and precisely handle symbolic constants in the predicates.

4.1 Preliminaries

It is useful in the following to assume that programs are in *internal form*. A program P is in internal form if each procedure of P has the following properties:

—The procedure has a single **return** statement, and it is the last statement of the procedure. Further, that statement has the form **return** `r`, where `r` is a local or formal of the procedure.

—Each statement in the procedure has a unique label.

—The actual parameters of each procedure call in the procedure are variables, rather than arbitrary expressions.

—The left-hand side of each procedure call statement has only local variables.

—For each formal parameter f of the procedure, there is a local variable $'f$ and an *initialization statement* $L : 'f := f$ at the beginning of the procedure's sequence of statements. The variable $'f$ is never referenced again in the procedure. This requirement initializes symbolic constants to their appropriate values.

It is clear that transforming a program to internal form does not change its semantics. Throughout the rest of the section, we assume programs are in internal form.

Let $G_P = \{g_1, g_2, \ldots\}$ be the global variables of the program P. For a procedure R, let $F_R = \{f_1, f_2, \ldots\}$ be the formal parameters of R, and let $L_R = \{l_1, l_2, \ldots\}$ be the local variables of R.

Each predicate in $E = \{\varphi_1, \ldots, \varphi_n\}$ will have a corresponding b-variable in $BP(P, E)$. The user annotates each predicate in E as being either global to

[4]Technically, the **choose** function given in Section 2 is not in the language of Section 3 (because the language does not contain an `if-then-else` statement). We also often employ a parallel assignment where the right-hand sides are all calls to **choose**, another violation of the language's syntax. Both of these relaxations can be straightforwardly desugared to the appropriate syntax.

$BP(P, E)$ or local to a particular procedure in $BP(P, E)$ (see Figure 1, in which predicates are declared local to `inc` or `foo`—there are no global predicates declared in that example), thereby determining the scope of the corresponding b-variable in $BP(P, E)$. Let E_G denote the global predicates of E. Global predicates must only reference variables in G_P. $BP(P, E)$ will contain one global b-variable declaration for each predicate in E_G. For a procedure R, let E_R denote the subset of predicates in E that are local to R. Local predicates to R can reference only variables in scope of R. $BP(P, E)$ will contain either a local b-variable declaration or a formal b-parameter declaration in its abstraction of R for each predicate in E_R.

Let $V = \{b_1, \ldots, b_n\}$ be the associated b-variables of the predicates in E and let $\mathcal{E}$ be a mapping from each b_i to the associated φ_i. We often extend $\mathcal{E}$ to negations, conjunctions, and disjunctions of Boolean variables, in the obvious way.

4.2 Weakest Preconditions and Monomials

For a statement s and a predicate φ, let $WP(s, \varphi)$ denote the *weakest liberal precondition* [Dijkstra 1976; Gries 1981] of φ with respect to statement s. $WP(s, \varphi)$ is defined as the weakest predicate whose truth before s entails the truth of φ afterward. The standard weakest precondition rule says that $WP(x := e, \varphi)$ is φ with all occurrences of x replaced with e, denoted $\varphi[e/x]$. For example, $WP(\texttt{x:=x+1},\ x < 5) = (x + 1) < 5 = (x < 4)$. Therefore, $(x < 4)$ is true before `x:=x+1` executes if and only if $(x < 5)$ is true afterward.

Given a statement s, a set of predicates E, and predicate $\varphi \in E$, it may be the case that $WP(s, \varphi)$ is not in E. For example, suppose $E = \{(x < 5), (x = 2)\}$. We have seen that $WP(\texttt{x:=x+1},\ x < 5) = (x < 4)$, but the predicate $(x < 4)$ is not in E. Therefore, we use decision procedures (i.e., a theorem prover) to *strengthen* the weakest precondition to an expression over the predicates in E. In our example, we can show that $x = 2 \Rightarrow x < 4$. Therefore, if $(x = 2)$ is **true** before `x:=x+1`, then $(x < 5)$ is **true** afterward.

We formalize this strengthening of a predicate as follows: A *monomial* over a set of Boolean variables $V = \{b_1, \ldots, b_n\}$ is a conjunction $c_1 \wedge \cdots \wedge c_n$, where each $c_i \in \{b_i, \neg b_i\}$. For any predicate φ, a set of Boolean variables V, and a function $\mathcal{E}$ that maps each variable b in V to a predicate $\mathcal{E}(b)$ in E, let $\mathcal{F}_{V,\mathcal{E}}(\varphi)$ denote the disjunction of all monomials m over V such that $\mathcal{E}(m)$ implies φ. The predicate $\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(\varphi))$ represents the weakest predicate over E that is stronger than φ. In our example, if $V = \{b_1, b_2\}$, $\mathcal{E}(b_1) = (x < 5)$, and $\mathcal{E}(b_2) = (x = 2)$, then $\mathcal{F}_{V,\mathcal{E}}(x < 4) = (b_1 \wedge b_2) \vee (\neg b_1 \wedge b_2)$, so

$$\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(x < 4)) = ((x < 5) \wedge (x = 2)) \vee (\neg(x < 5) \wedge (x = 2)),$$

which is equivalent to $(x = 2)$.

It will also be useful to define a corresponding weakening of a predicate to the set of predicates in E. Define $\mathcal{G}_{V,\mathcal{E}}(\varphi)$ as $\neg\mathcal{F}_{V,\mathcal{E}}(\neg\varphi)$. The predicate $\mathcal{E}(\mathcal{G}_{V,\mathcal{E}}(\varphi))$ represents the strongest predicate over E that is implied by φ.

For each monomial, the implication check involves a call to a theorem prover implementing the required decision procedures. Our implementation of C2BP

uses Simplify [Detlefs et al. 2003], a Nelson–Oppen theorem prover [Nelson 1981]. The naive computations of $\mathcal{F}_{V,\mathcal{E}}(\cdot)$ and $\mathcal{G}_{V,\mathcal{E}}(\cdot)$ require exponentially many calls to the theorem prover in the worst case. We have implemented several optimizations that make the $\mathcal{F}_{V,\mathcal{E}}(\cdot)$ and $\mathcal{G}_{V,\mathcal{E}}(\cdot)$ computations practical [Ball et al. 2001a].

4.3 Abstracting Statements Besides Procedure Calls and Returns

Each statement of program P (other than a procedure call or return) abstracts to one statement in $BP(P, E)$. We define the function $BP(s, V, \mathcal{E})$ to output the abstraction of statement s, given the set of Boolean variables V and a mapping $\mathcal{E}$ to the associated predicates being modeled. Most of the statement abstractions are straightforward:

$$\begin{aligned} BP(\mathbf{skip}, V, \mathcal{E}) &\equiv \mathbf{skip} \\ BP(\mathbf{goto\,L}, V, \mathcal{E}) &\equiv \mathbf{goto\,L} \\ BP(\mathbf{L}\colon s, V, \mathcal{E}) &\equiv \mathbf{L}\colon BP(s, V, \mathcal{E}) \\ BP(\mathbf{branch}\ \overline{s_1}\|\cdots\|\overline{s_n}\ \mathbf{end}, V, \mathcal{E}) &\equiv \mathbf{branch}\ BP(\overline{s_1}, V, \mathcal{E})\|\cdots\|BP(\overline{s_n}, V, \mathcal{E})\ \mathbf{end} \end{aligned}$$

The notation $BP(\overline{s}, V, \mathcal{E})$ used in the abstraction for **branch** statements abbreviates

$$BP(s_1, V, \mathcal{E}); \cdots BP(s_n, V, \mathcal{E});$$

where $\overline{s} = s_1; \cdots s_n;$.

For a statement s of the form **assume**(e), we know by the semantics of **assume** that e is true immediately after the execution of s in P. However, it is possible that e is not in the set E of predicates being modeled, so the best we can do in $BP(P, E)$ is to assume the strongest predicate over expressions in E that is implied by e:

$$BP(\mathbf{assume}(e), V, \mathcal{E}) \equiv \mathbf{assume}(\mathcal{G}_{V,\mathcal{E}}(e))$$

For example, suppose the statement s is **assume**`(x < 2)` and the set of predicates E is $\{(x < 5), (x = 2)\}$. Then $BP(s, V, \mathcal{E})$ (under the obvious V and $\mathcal{E}$ interpretations) is **assume**`({x<5}&!{x=2})`.

Finally, consider an assignment statement s of the form $x := e$.[5] The associated statement in $BP(P, E)$ must appropriately update all of the Boolean variables in V that are in scope at statement s. For every b_i, if $WP(s, \varphi_i)$ is true before execution of s, then b_i may be safely set to true in $BP(P, E)$. Similarly, if $WP(s, \neg\varphi_i)$ is true before execution of s, then b_i may be safely set to false in $BP(P, E)$. Because the predicate $WP(s, \varphi_i)$ may not be in E, we need to strengthen it to a predicate over expressions in E that implies $WP(s, \varphi_i)$, and similarly for $WP(s, \neg\varphi_i)$. Therefore, $BP(P, E)$ will contain the following parallel

[5]The abstraction process can be straightforwardly extended to handle parallel assignments in the source program.

assignment statement in place of s[6]:

$$\begin{aligned} BP(x := e, V, \mathcal{E}) \quad \equiv \quad & b_1, \ldots, b_n := \\ & \texttt{choose}(\mathcal{F}_{V,\mathcal{E}}(WP(x := e, \varphi_1)), \mathcal{F}_{V,\mathcal{E}}(WP(x := e, \neg\varphi_1))), \\ & \ldots, \\ & \texttt{choose}(\mathcal{F}_{V,\mathcal{E}}(WP(x := e, \varphi_n)), \mathcal{F}_{V,\mathcal{E}}(WP(x := e, \neg\varphi_n))). \end{aligned}$$

Consider again the assignment statement `x:=x+1` and the set of predicates $E = \{(x < 5), (x = 2)\}$. The abstraction of this assignment in the Boolean program is:

```
{x<5}, {x=2} := choose({x=2}, !{x<5}), choose(false, {x=2}|!{x<5});.
```

4.4 Abstracting Procedures, Calls, and Returns

This section describes the abstraction of procedures and procedure calls. When abstracting a program P, C2BP first produces the *interface* of each procedure in P, which is essentially the procedure's type signature in $BP(P, E)$. Interfaces can be determined for each procedure in isolation. Once this is done, the statements of each procedure are abstracted one-by-one; the abstraction of a call to procedure R relies only on R's interface.

4.4.1 *Procedure Interfaces.* Let R' be the version of procedure R in $BP(P, E)$. The interface of R' is a six-tuple $(F_R, r, S_R, E_f, E_r, B_R)$, which C2BP constructs by examining E_R and the procedure R in program P. The terms F_R and r (the variable returned by R) were defined earlier. $S_R = \{'f_1, 'f_2, \ldots\}$ is the set of symbolic constants used in the predicates in E_R.

Let $scs(\varphi)$ be the set of symbolic constants in predicate φ. Let $vars(\varphi)$ be the set of variables referenced in predicate φ. The set E_f contains the subset of E_R that should be formal parameter predicates in $BP(P, E)$. Intuitively, these are the predicates that can be given a meaning in the calling context before R executes. Specifically, these are the predicates in E_R that refer to variables in F_R, and possibly to globals, but do not refer to local variables of R or to symbolic constants[7]

$$E_f = \{\varphi \in E_R \mid vars(\varphi) \cap F_R \neq \emptyset \wedge vars(\varphi) \cap L_R = \emptyset \wedge scs(\varphi) = \emptyset\}.$$

The associated b-variable in V of each member of E_f will be a formal b-parameter in R'. The associated b-variables of all other members of E_R will be declared as local variables in R'. For example, in Figure 3, `{a=2}` is a formal b-parameter because it only refers to the formal a of `foo`, while `{b=3}` and `{c=4}` are local b-variables because b and c are locals.

The set E_r contains the subset of predicates in E_R that should be returned by R'. Intuitively, these are the predicates that can be given a meaning in the calling context after R executes. Specifically, these are the predicates in E_R

[6]This "pointwise" updating of the Boolean variables corresponds to a Cartesian approximation of the most precise Boolean abstraction possible. For details, see Ball et al. [2001b].

[7]Although symbolic constants can be given a meaning in the calling context, it is not useful to include them, because a symbolic constant will have the same value as its corresponding formal parameter initially.

that refer to the return variable r or to symbolic constants, but not to local variables or formals of R (other than r):

$$E_r = \{\varphi \in E_R \mid (r \in vars(\varphi) \vee scs(\varphi) \neq \emptyset) \wedge (vars(\varphi) - \{r\}) \cap (L_R \cup F_R) = \emptyset\}.$$

For example, in Figure 3 both local b-variables of `inc` are returned, because x is the return variable of `inc` in the original C program and $'x$ is a symbolic constant.

Finally, for each symbolic constant $'f$ in S_R, we require that there exist a predicate $(f = 'f)$ in E_R.[8] The set $B_R \subseteq E_R$ consists of the predicates in E_R of the form $(f = 'f)$, which are called the *binding predicates* of R. They are used, in conjunction with the initialization statements required by internal form, to provide the Boolean program abstraction with the information that initially each symbolic constant has the same value as its corresponding formal parameter. Procedure `inc` in Figure 3 has a single binding predicate $(x =' x)$. Its corresponding b-variable is initially set to true as a result of performing the ordinary abstraction for assignment statements (discussed earlier) on the initialization statement `'x := x` in the internal form of `inc` in Figure 1.

4.4.2 *Abstracting Return Statements.* Now we can define the rule for abstracting **return** statements. Let $b_1, \ldots, b_m$ be the associated b-variables in V for each member of E_r. Then the abstraction of R's **return** statement is defined as follows:

$$BP(\textbf{return}\ r,\ V, \mathcal{E}) \equiv \textbf{return}\ b_1, \ldots, b_m$$

4.4.3 *Abstracting Procedure Calls.* Consider a call $v := R(a_1, \ldots, a_j)$ in some procedure Q in program P.[9] Let V_Q be the b-variables in scope at Q' (the Boolean program version of Q), and let $\mathcal{E}_Q$ be the submap of $\mathcal{E}$ from V_Q to the predicates they represent. Let $(F_R, r, S_R, E_f, E_r, B_R)$ be the interface of R'. We divide the computation of $BP(v := R(a_1, \ldots, a_j), V_Q, \mathcal{E}_Q)$ into three parts: computation of the actual parameters, generation of the call to R', and updating of the variables in the calling context.

First, C2BP computes an actual value to pass to R' for each formal parameter predicate $\varphi \in E_f$, assigning it to a fresh local variable *prm*. Let $\varphi' = \varphi[a_1/f_1, a_2/f_2, \ldots, a_j/f_j]$, where $f_1, f_2, \ldots, f_j$ are the formals from F_R. The expression φ' represents the value of formal parameter predicate φ in the current calling context. Therefore, if φ' is true (false) before the call, then the corresponding formal b-parameter of φ in R' should be passed the value true (false). As with assignment statements, in general we need to strengthen φ' to an expression over the expressions in E:

$$prm := \texttt{choose}(\mathcal{F}_{V_Q,\mathcal{E}_Q}(\varphi'), \mathcal{F}_{V_Q,\mathcal{E}_Q}(\neg\varphi')); \tag{1}$$

As an example, consider the abstraction of the call `b := inc(a)` in Figure 2. The b-variables V_{foo} in scope at `foo` are {{`a=2`}, {`b=3`}, {`c=4`}}. The $\mathcal{E}_{foo}$ function is obvious from the notation (e.g., $\mathcal{E}_{foo}(\{\texttt{a=2}\}) = (a = 2)$, etc.) The formal

[8] A tool could easily insert such predicates into E_R as necessary.

[9] For simplicity, we assume that the call has a single return value. Extending the abstraction process to multiple return values is straightforward.

b-parameters of `inc` in the abstraction are {x=2}, {x=3} and {x=4}. In order to compute the actual for {x=2}, C2BP outputs

$$\texttt{prm1} := \texttt{choose}(\mathcal{F}_{V_{foo},\mathcal{E}_{foo}}(a = 2), \mathcal{F}_{V_{foo},\mathcal{E}_{foo}}(a \neq 2)),$$

where $(a = 2)$ is obtained by substituting the actual `a` for formal `x`. The values for `prm2` and `prm3` are computed in a similar manner.

Generating the call to R' is straightforward: we pass all $|E_f|$ of the *prm* variables to R', and we use $|E_r|$ fresh *ret* variables to catch the return values from the call:

$$ret_1, \ldots, ret_{|E_r|} := R'(prm_1, \ldots, prm_{|E_f|}); \tag{2}$$

The most difficult part of the abstraction is in conservatively (but precisely) updating the b-variables in the scope of Q' whose associated predicates can change values as a result of a call to R. In particular, any predicate that mentions v (the left-hand side of the call to R in Q) or a global variable might have changed its value. The set of such predicates is formally defined as follows:

$$E_u = \{\varphi \in E_Q \mid (v \in vars(\varphi)) \vee (vars(\varphi) \cap G_P \neq \emptyset)\}.$$

We will update the value of each b-variable whose corresponding predicate is in E_u, in the context of the return predicates of R' as well as the old values of the b-variables in V_Q. To perform this update, it is useful to conceptually consider the call $v := R(a_1, \ldots, a_j)$ equivalently as $v_{ret} := R(a_1, \ldots, a_j); v := v_{ret}$, where v_{ret} is a fresh variable. This allows us to capture the intermediate state when the call from R has returned but before the update to v, which is useful for the generalization to handle pointers in Section 5. We saw above that for each return predicate φ of E_r, there is an associated b-variable *ret* local to procedure Q'. Let T_Q be the set of these b-variables and let $\hat{V} = V_Q \cup T_Q$. The effect of the implicit statement $v_{ret} := R(a_1, \ldots, a_j)$ is accounted for by the creation of a temporary map $\hat{\mathcal{E}}$ that provides a conservative interpretation for each b-variable in $\hat{V}$ after the call (but before the update to v). This map is then used to update the corresponding b-variable b of each predicate φ in E_u, with respect to the implicit assignment $v := v_{ret}$:

$$b := \texttt{choose}(\mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP(\texttt{v} := \texttt{v}_{ret}, \varphi)), \mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP(\texttt{v} := \texttt{v}_{ret}, \neg\varphi))); \tag{3}$$

To complete the description of the algorithm, we provide the definition of $\hat{\mathcal{E}}$. We distinguish three categories of b-variables in $\hat{V}$: globals in V_Q, locals and formals in V_Q, and the variables in T_Q. Recall that $\mathcal{E}_Q$ maps the b-variables V_Q in scope in Q to predicates that they represent. Because the globals in V_Q are also in scope of R' (by nature of being globals), their truth values will be updated properly by the call to R'. Therefore, their ordinary interpretation suffices:

$$\hat{\mathcal{E}}(b) = \mathcal{E}_Q(b) \text{ if } b \text{ is a global } b\text{-variable in } V_Q.$$

The truth values of locals and formals in V_Q may have been invalidated by the call. In particular, such b-variables are not in scope of R' (and hence are not modified by R') but their associated predicates may depend on the value of a global variable in the original C program, whose value can be modified by a call to R in P. To address this discrepancy, we introduce a mapping *orig* that says

how to interpret each variable in these predicates after the call to R. For local variables and formal parameters y of Q, we define $orig(y) = y$, since their values are not affected by the call to R (as there are no references, for now). For global variables y, we define $orig(y) = y_o$, where y_o is a fresh variable called an *original variable*. Conceptually, we can think of y_o as representing the unknown value of the global variable y before the call to R. We extend *orig* to expressions in the usual way. Then we define $\hat{\mathcal{E}}$ as follows for local and formal b-variables of V_Q:

$$\hat{\mathcal{E}}(b) = orig(\mathcal{E}_Q(b)) \text{ if } b \text{ is a local or formal } b\text{-variable in } V_Q$$

For example, consider the first call to `inc` in procedure `foo` of Figure 3. After this call, $\hat{\mathcal{E}}$ maps the b-variable`{a=2}` to $orig(a = 2)$, which is simply $(a = 2)$. Because a is a local variable, its value cannot be affected by the call. Therefore, if `{a=2}` is true after the call to `inc` in the Boolean program, we are assured that $(a = 2)$ is true after the call to `inc` in the original program. However, if a were a global variable (but $(a = 2)$ were still declared a local predicate of `foo`), then `{a=2}` would instead be mapped to $(a_o = 2)$, to account for the fact that a could be modified by the call to `inc` in P. Therefore, if `{a=2}` is true after the call to `inc` in the Boolean program, we may only assume that the original, unknown value of a, rather than its current value, is equal to 2 in the original program.

Lastly, we consider the interpretation of b-variables in T_Q. We saw above that for each return predicate φ of E_r for R', there is an associated b-variable *ret* local to procedure Q'. Let $\mathcal{E}_{RQ}$ be a map with domain T_Q such that $\mathcal{E}_{RQ}(ret) = \varphi$ for each such pair. We define a mapping γ which says how to interpret the return predicates in the calling context. Since a symbolic constant $'f$ refers to the initial value of a formal parameter f of R, we define $\gamma('f) = orig(a)$, where a is the associated actual of f in the call to R. For the return variable r in R, $\gamma(r) = v_{ret}$. Finally, since R appropriately updates globals, for each global g in G_P we have $\gamma(g) = g$. We extend γ to expressions in the usual way. Then, we can define $\hat{\mathcal{E}}$ as follows for the variables in T_Q:

$$\hat{\mathcal{E}}(b) = \gamma(\mathcal{E}_{RQ}(b)) \text{ if } b \in T_Q.$$

Again consider the first call to `inc` in procedure `foo` of Figure 3. After this call, $\hat{\mathcal{E}}$ maps the b-variable `ret1` to $\gamma(x = 'x) = (b_{ret} = a)$.

Putting all of the pieces together, we have:

$$\hat{\mathcal{E}}(b) = \begin{array}{ll} orig(\mathcal{E}_Q(b)), & \text{if } b \text{ is a local or formal } b\text{-variable in } V_Q \\ \mathcal{E}_Q(b), & \text{if } b \text{ is a global } b\text{-variable in } V_Q \\ \gamma(\mathcal{E}_{RQ}(b)), & \text{if } b \in T_Q. \end{array} \tag{4}$$

Consider the processing of return values from the abstraction of the call `b := inc(a)` in the monomorphic predicate abstraction of Figure 2. The set of return temporaries T_{foo} is {`ret1`, `ret2`, `ret3`}. We have $\hat{V} = V_{foo} \cup T_{foo}$, $\hat{\mathcal{E}}$(`ret1`) $= (b_{ret} = 2)$, $\hat{\mathcal{E}}$(`ret2`) $= (b_{ret} = 3)$, and $\hat{\mathcal{E}}$(`ret3`) $= (b_{ret} = 4)$, obtained by substituting b_{ret} for x in the return predicates of `inc`. We also have $\hat{\mathcal{E}}(\{$`a=2`$\}) = (a = 2)$, $\hat{\mathcal{E}}(\{$`b=3`$\}) = (b = 3)$, and $\hat{\mathcal{E}}(\{$`c=4`$\}) = (c = 4)$. The new value of `{b=3}` is `choose`$(\mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP($`b :=` $\mathtt{b_{ret}}, b = 3)), \mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP($`b :=` $\mathtt{b_{ret}}, b \neq 3))$, which compiles to `choose(ret2, !ret2)`.

Now consider the abstraction of the call `b := inc(a)` in the polymorphic predicate abstraction of Figure 3. The b-variables V_{foo} in scope at `foo` are {{a=2}, {b=3}, {c=4}}, with the same interpretation as above. The set of return temporaries T_{foo} is {ret1, ret2}. We have $\hat{\mathcal{E}}(\texttt{ret1}) = (b_{ret} = a)$ and $\hat{\mathcal{E}}(\texttt{ret2}) = (b_{ret} = a + 1)$. The new value of {b=3} is $\texttt{choose}(\mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(b_{ret} = 3), \mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(b_{ret} \neq 3))$, which compiles to

```
choose({a=2}&ret2, ({a=2}&!ret2)|!{a=2}&ret2))
```

To summarize, $BP(v := R(a_1, \ldots, a_j), V_Q, \mathcal{E}_Q)$ is obtained by concatenating the assignments of the form (1) for every predicate in E_f, a call of the form (2) and assignments of the form (3) for every predicate in E_u.

5. ADDING POINTERS

In this section, we extend the algorithm of Section 4 to handle programs with pointers. We now allow the full syntax of the language of Section 3. We also now allow pointers, pointer dereferencing, and addresses of variables in the predicates in E. In addition, symbolic constants may now refer to dereferences of formal parameters. Define an *access expression* q to be a variable preceded by zero or more dereference (*) symbols. Let $var(q)$ be the underlying variable in the access expression. Consider a procedure R. Given an access expression q, where $var(q)$ is a formal parameter of R, the symbolic constant $'q$ refers to the value of q on entry to R. These generalizations require modification to the internal form as well as the translation of assignment statements, procedure interfaces, and procedure calls.

5.1 Internal Form

We modify the definition of the internal form in two ways. First, we allow expressions of the form $\&x$ and $*^n x$ (the value of $n \geq 0$ dereferences of x) as actual parameters of calls, in addition to variables. Second, we augment initialization statements to properly initialize symbolic constants of dereferences to formal parameters. For each procedure, for each access expression $*^n f$ such that f is a formal of the procedure and the access expression $*^n f$ is allowed by the formal's type, we require a local variable $'*^n f$. Further, we require the existence of an initialization statement $L : '*^n f := *^n {}' f$ after the initialization statement for $' f$. The variable $'*^n f$ is never referenced again in the procedure.

5.2 Assignment Statements

When translating an assignment statement (either of the form $x := e$ or $*x := e$), the standard weakest precondition computation described in Section 4.2 no longer suffices. For example, consider $WP(\texttt{x} := 5, *y > 6)$. According to the standard definition, $WP(\texttt{x} := 5, *y > 6) = (*y > 6)[5/x] = *y > 6$. This means that if $*y > 6$ is true before the execution of `x:=5`, then it is true afterward. However, this is not the case if $*y$ and x are aliases of one another.

To handle this problem, we adapt Morris' general axiom of assignment [Morris 1982; Ball et al. 2001a]. Let metavariable l range over *locations*, which are l-valued expressions. Consider the computation of $WP(\texttt{l} := \texttt{e}, \varphi)$, and let l'

be a location mentioned in the predicate φ. Then there are two cases to consider: either l and l' are aliased, and hence a change in the contents of l will cause a change in the contents of l'; or they are not, and an assignment to l leaves l' unchanged. Define

$$l'[l \leftarrow e] = \begin{cases} e & \text{if } l \text{ and } l' \text{ are aliased;} \\ l' & \text{otherwise.} \end{cases}$$

Then the predicate $WP(\texttt{l:=e}, \varphi)$ is defined as $\varphi[l \leftarrow e]$, where $\varphi[l \leftarrow e]$ denotes the predicate obtained by syntactically substituting each location l' in φ by the expression $l'[l \leftarrow e]$. In this way, all aliases of l are replaced by e.

Of course, this ideal weakest precondition semantics is not statically computable in general, because we do not have complete alias information. Instead, the weakest precondition that we compute explicitly considers all possible alias scenarios for l. If φ has k locations in it, there will be 2^k alias scenarios to consider, since l could be aliased to any subset of these k locations. For each scenario, we compute $\varphi[l \leftarrow e]$, simultaneously substituting all locations in φ that are aliased to l under the current alias scenario with e. The resulting predicate is conjoined with a conjunction of equality predicates that formally describe the associated alias scenario. Finally, the complete weakest precondition is the disjunction of the predicates computed for each alias scenario. In our example above, $WP(\texttt{x := 5}, *y > 6) = ((\&x = y) \wedge (5 > 6)) \vee ((\&x \neq y) \wedge (*y > 6))$. We use a may-alias analysis [Das 2000] to improve the precision of this weakest precondition computation, pruning disjuncts that represent infeasible alias scenarios [Ball et al. 2001a].

5.3 Procedure Interfaces

The components of a procedure R's interface are defined as in Section 4.4, with one exception. The definition of binding predicates B_R is augmented to provide the Boolean program with information about the values of the new kinds of symbolic constants. We require predicates of the form $(*^n\,'f = {}'*^n f)$ where ${}'*^n f$ is in S_R.

5.4 Procedure Calls

Consider a call $v := R(a_1, \ldots, a_j)$ in some procedure Q in program P. The computation of the actuals for the corresponding call to R' in $BP(P, E)$ is unchanged, as is the generation of the call to R'. However, in the presence of pointers, the set of local predicates E_u in Q' that must be updated after the call to R needs to be generalized to include predicates that can be invalidated due to pointer indirection and aliasing. Let l be a location in scope in Q. We say that l is *possibly affected* by the call to R if before the call any of the following conditions holds:

—l is an alias of a global variable

—l is an alias of a (transitive) dereference of a global variable

—l is an alias of a (transitive) dereference of an actual parameter to the call to R

The set of locations that are possibly affected by the call to R is computed conservatively using a may-alias analysis. Then E_u can be conservatively defined as follows:

$$E_u = \{\varphi \in E_Q \mid \varphi \text{ references an alias of } v \ \vee \\ \varphi \text{ references a location that is possibly affected by the call to } R\}.$$

Next, the mappings *orig* and γ are extended. The mapping *orig* is generalized to handle all locations. Given an access expression $*^n y$, if that access expression is possibly affected by the call to R, then $orig(*^n y) = y_{o,n}$, where $y_{o,n}$ is fresh. As before, $y_{o,n}$ is used to represent the unknown value of $*^n y$ before the call to R. Otherwise, $orig(*^n y) = *^n y$. Also, $orig(\&y) = \&y$. We sometimes abbreviate $y_{o,0}$ as y_o.

Now γ is generalized in the natural way to use the extension of *orig*. Specifically, if f is a formal parameter of R with associated actual a, then

$$\gamma('*^n f) = orig(*^n a).$$

For example, consider the following simple procedure:

```
void simple(int* p) {
  'p := p;
  '*p := *'p;
  *p := 8;
}.
```

The first two assignments are the initialization statements. The symbolic constant $'*p$ refers to the value of $*p$ at the entry of the procedure. On the other hand, $*'p$ refers to the value obtained by dereferencing the original value of p. Note that unlike $'*p$, the value of $*'p$ can change through the procedure. For example, after the assignment to $*p$, $*'p$ has the value 8, while $'*p$ still has the original value of $*p$. Assume a is the actual parameter to some call to `simple`, and assume a is a local variable of its procedure. Then γ maps $'*p$ to $a_{o,1}$ at that call site, to represent the unknown value of $*a$ before the call. On the other hand, γ maps $*'p$ to $*a$, which has the updated value 8 in the calling context.

Finally, updating the predicates in E_u proceeds as described in Section 4.4. In particular, the extended *orig* and γ mappings are used in the definition of $\hat{\mathcal{E}}$ shown in (4). The extended $\hat{\mathcal{E}}$ mapping in turn is used to update the b-variables corresponding to predicates in E_u, as shown in (3).

Appendix B contains the proof of soundness of the full C2BP algorithm including pointers.

5.5 Example

Figure 5 shows a detailed example involving pointers. The polymorphic predicates in the `swap` function allow all callers to prove that the values of $*p$ and $*q$ are swapped after the call to `swap` returns. Figure 6 shows the (optimized) Boolean program output by C2BP.

The `swap` function has four binding predicates, $(p = 'p)$, $(q = 'q)$, $(*'p = '*p)$ and $(*'q = '*q)$, which are initialized to true at the beginning of the procedure as a result of abstracting the initialization statements introduced by the

```
void
swap(ref int p, ref int q) {
  int t;
  t := *p;
  *p := *q;
  *q := t;
  return;
}

void test() {
  int x, y;
  x := 5;
  y := 4;
  swap(&x,&y);
  return;
}
```

(a) **Program**

```
swap {
  p = 'p,
  q = 'q,
  *'p = '*p,
  *'q = '*q,
  *'p = '*q,
  *'q = '*p,
  t = '*p
}

test {
  x = 4,
  x = 5,
  y = 4,
  y = 5
}
```

(b) **Predicates**

Fig. 5. Polymorphic predicate abstraction with pointers

```
bool,bool,bool,bool swap() {
  bool {p='p},{q='q};
  bool {*'p='*p},{*'q='*q},
       {*'p='*q},{*'q='*p};
  bool {t='*p};

  {p='p} := true;
  {q='q} := true;
  {*'p='*p} := true;
  {*'q='*q} := true;

  {t='*p} := {p='p} & {*'p='*p}; // t := *p;

  //*p := *q;
  {*'p='*q}, {*'p=*'p} :=
     choose({p='p}&{q='q}&{*'q='*q},
            {p='p}&{q='q}&!{*'q='*q}),
     choose({p='p}&{q='q}&{*'q='*p},
            {p='p}&{q='q}&!{*'q='*p});

  //*q := t;
  {*'q='*p},{*'q='*q} :=
     choose({q='q}&{t='*p},
            {q='q}&!{t='*p}), *;

  return {*'p='*p},{*'q='*q},
         {*'p='*q},{*'q='*p};
}
```

```
void test() {
  bool {x=4},{x=5},{y=4},{y=5};
  bool ret1,ret2,ret3,ret4;

  {x=5},{x=4} := true, false; // x := 5;
  {y=4},{y=5} := false, true; // y := 4;

  ret1,ret2,ret3,ret4 := swap();

  {x=5},{x=4},{y=5},{y=4} :=
    choose({y=5}&ret3|{x=5}&ret1, ...),
    choose({y=4}&ret3|{x=4}&ret1, ...),
    choose({y=5}&ret2|{x=5}&ret4, ...),
    choose({y=4}&ret2|{x=4}&ret4, ...);
  return;
}
```

Fig. 6. Boolean program abstraction created by the C2BP tool, given the input program and predicates shown in Figure 5.

internal form. The abstraction of the three assignment statements uses the updated weakest precondition for pointers. For example, when abstracting the statement `*q := t`, we must consider the possibility that $*q$ is aliased to $*p$, in which case predicates involving $*p$ (and its aliases) may be affected. In this case, however, our (global) alias analysis deduces that $*p$ and $*q$ are not aliased, since the actuals for p and q in the sole call to `swap` are distinct addresses $\&x$ and $\&y$.

According to the definition of E_r, the predicates $(*'p = '*p), (*'q = '*q)$, $(*'p = '*q)$, and $(*'q = '*p)$ are returned from `swap`. Consider the call to `swap` from the `test` function. Because x and y are dereferences of the actual parameters to the call, they are both possibly affected by the call. Therefore, all four local predicates of `test` are in E_u as defined earlier, so they must be updated after the call. By the γ mapping, the returned predicates $\{(*'p = '*p), (*'q = '*q), (*'p = '*q), (*'q = '*p)\}$ are mapped to the predicates $\{(x = x_o), (y = y_o), (x = y_o), (y = x_o)\}$.[10] For example, $\gamma('*p) = orig(x) = x_o$, representing the unknown value of x before the call. Similarly, the local predicates $\{(x = 4), (y = 5), (x = 5), (y = 4)\}$ are mapped by *orig* to $\{(x_o = 4), (y_o = 5), (x_o = 5), (y_o = 4)\}$. These mappings suffice to prove that the swap property holds in `test`. For example, we can deduce that if $(y_o = 4)$ and $(x = y_o)$ are true, then $(x = 4)$ is also true after the call to `swap`. This can be seen by the update to `{x=4}` after the call to `swap` in the abstraction of `test`.

6. RELATED WORK

Predicate abstraction was first introduced by Graf and Saïdi [1997] in the context of guarded transition systems. Others have since reported on variants and optimizations of the original technique for similar kinds of transition systems [Das et al. 1999; Saïdi and Shankar 1999; Clarke et al. 2000]. The transition systems considered in these works lack important source-level features including pointers and procedures, making them unsuitable for representing software directly. Instead, a transition system is typically used to represent a model of some software protocol, and that model is then abstracted and validated.

More recently, predicate abstraction has been applied directly to source-level languages. Visser et al. [2000] describe two systems for validating properties of software. The first system model checks C++ code by translating it to Promela, the input language of the Spin model checker [Holzmann 1997]. Predicate abstraction is applied manually in order to drastically reduce the state space of the resulting models, thereby speeding up verification considerably. The second system is an automatic predicate abstractor for Java. The output of the abstraction process is another Java program, with some of the original variables removed and with Boolean variables added to track user-specified predicates. This Java program can then be validated with a model checker such as the Java Pathfinder [Havelund and Pressburger 1998]. Because Java is the target language of the predicate abstraction, pointers can be retained in the translation, rather than carefully abstracted away as in our approach. For a similar

[10]Technically, x should be $*\&x$, and similarly for y.

reason, dynamic memory allocation and concurrency are handled, while safely and precisely supporting these features in C2BP would be quite challenging. On the other hand, Boolean programs are simple enough to support a decidable model checking algorithm, while Java model checking will not terminate in general. Therefore, in practice, a Java model checker may be able to find bugs but not prove their absence. It is not clear how the described predicate abstractor for Java handles procedure calls, and polymorphic predicates are not supported.

The BLAST tool [Henzinger et al. 2002] is, like SLAM, a software model checker for C programs. The key innovation of BLAST is the idea of *lazy abstraction*. Rather than treating abstraction, model checking, and predicate discovery as three independent tasks, the BLAST tool integrates them in order to reduce redundant and unnecessary work. Abstraction is performed on demand during a depth-first exploration of the program's state space. When an error state is reached, new predicates are added, but only on the portion of the state space that can lead to that error state. Exploration of a path terminates upon either reaching an error state or reaching a state that has already been explored in the same (or a weaker) context. Unlike SLAM's modular abstraction process, BLAST handles procedure calls by inline-expansion, abstracting and model checking a procedure body anew in different contexts. Based on our work, BLAST has recently been extended to support polymorphic predicates [Henzinger et al. 2004]. Polymorphic predicates allow BLAST to analyze a procedure body fewer times during model checking, by finding more calling contexts to be similar to one another. Polymorphic predicates are also employed in the predicate discovery phase to compute an *interpolant*, which is used to reduce the number of new predicates that must be added to the abstraction in the next iteration.

Flanagan and Qadeer [2002] show how to use predicate abstraction to infer loop invariants for Java programs, in the context of the ESC/Java system [Flanagan et al. 2002]. The programmer annotates a loop with predicates of interest, and predicate abstraction is used to synthesize the strongest loop invariant that can be expressed as a Boolean combination of those predicates. The technique allows *skolem constants* in predicates, which are used to infer universally quantified loop invariants. Skolem constants play a similar role for loops that our symbolic constants play for procedures, allowing invariants to be deduced with respect to all possible contexts. The key difficulty of symbolic constants in our setting is the need to map them to each call site, so that callers can be safely and precisely abstracted. This mapping process has no analogue for loop invariants. In addition, our algorithm safely handles pointers to symbolic constants.

There have been several other proposals for automatically constructing abstract models of programs. Holzmann [2000] describes an approach to abstracting C code. The programmer defines an *abstraction table* that defines how various C constructs and code snippets should be treated in the abstract program. Holzmann's system uses this abstraction table to automatically construct a program's abstraction, which is then model checked with Spin. Similarly, Engler's meta-level compilation system [Engler et al. 2000] allows programmers to map

C-code patterns to abstract states. This mapping is then used to automatically determine whether a C program satisfies a user-specified finite-state protocol. The mapping technique enjoys a number of advantages. Users can express a wide range of useful abstractions. Further, because the abstraction mapping is completely user-directed, it is straightforward to accommodate complicated language features like concurrency. Finally, the abstraction algorithm is fairly lightweight and can scale to large systems. A disadvantage of the approach is that the user must direct the entire abstraction process. This contrasts with predicate abstraction, for which the user need only provide a set of predicates, from which the system automatically produces the abstraction for each program statement. Further, abstractions in the mapping approach are not guaranteed to be sound: a program may be buggy even if its abstraction is found to be bug-free. On the other hand, abstractions produced by C2BP are provably sound.

The Bandera toolset [Corbett et al. 2000; Dwyer et al. 2001] provides an automatic abstraction technique for Java programs based on abstract interpretation [Cousot and Cousot 1977]. For each type T in the program, the user provides an abstract domain of tokens and an abstraction function mapping each value of type T to its corresponding abstract token. The toolset uses this information to automatically produce sound abstractions of each operation of type T. The abstraction process then consists in replacing each concrete variable, literal, and operation with its corresponding abstract version. The resulting abstract Java program is model checked with Java PathFinder. Bandera's abstraction process is simpler than ours and can more easily scale to large programs. Also, as discussed earlier, using Java as the target language allows Bandera to handle features such as dynamic memory allocation and concurrency, at the expense of an undecidable model checking problem. The biggest drawback of Bandera's approach as compared with predicate abstraction is its limited expressiveness. Bandera's technique can be viewed as a special case of predicate abstraction in which each predicate mentions only a single variable. Predicate abstraction as embodied in C2BP is additionally able to capture interesting relationships among multiple variables.

The use of polymorphism in programming languages [Milner 1978; Cardelli and Wegner 1985] and program analysis has a rich history. For example, the ML programming language [Milner et al. 1997] has a polymorphic type system that allows the definition of generic functions, whose types contain *type variables*. For example, the identity function has type $\forall\alpha.\alpha \rightarrow \alpha$. A generic function can be safely typechecked once, no matter what types a particular calling context substitutes for the function's type variables. In our setting, symbolic constants allow for the generic abstraction of procedures. Each procedure can be safely abstracted once, no matter what values a particular calling context substitutes for the procedure's symbolic constants.

A *context-sensitive* program analysis is one that analyzes each call to a given procedure Q, based on Q's calling context [Sharir and Pnueli 1981]. *Transfer functions* summarize the input-output behavior of a procedure and provide a way to avoid redundant work during context-sensitive analysis [Sharir and Pnueli 1981]. The tools C2BP and BEBOP together compute transfer functions on demand [Ball et al. 2001a]. On-demand computation of transfer functions

$\wedge$	**true**	**false**	$*$
true	**true**	**false**	$*$
false	**false**	**false**	**false**
$*$	$*$	**false**	$*$

$\vee$	**true**	**false**	$*$
true	**true**	**true**	**true**
false	**true**	**false**	$*$
$*$	**true**	$*$	$*$

$\neg$	
true	**false**
false	**true**
$*$	$*$

Fig. 7. Kleene's three-valued interpretation of $\wedge$, $\vee$ and $\neg$.

has also been explored in the context of program analysis [Wilson and Lam 1995; Reps et al. 1995]. Polymorphic predicates allow us to raise the level of abstraction of the transfer functions computed in C2BP, allowing a single transfer function to be instantiated differently in different calling context. The use of polymorphism in transfer functions has also been explored in the context of work on polymorphic points-to analysis [Foster et al. 2000]. The binding of information between callers and callees has been discussed before in the context of interprocedural pointer analysis [Landi and Ryder 1992].

7. CONCLUSION

We have presented an algorithm for polymorphic predicate abstraction of C programs and proved it sound. Polymorphism is critical for obtaining the benefits of a modular abstraction process for procedures, allowing each procedure to have a single abstraction that can be safely and precisely used by all callers. In addition, polymorphism allows a procedure to be conservatively analyzed in an open environment, for all possible calling context. Finally, polymorphism can lead to more efficient abstraction, since a constant number of polymorphic predicates can often be used in place of monomorphic predicates proportional to the number of call sites of a procedure. The main technical challenge of polymorphic predicate abstraction is in capturing the effect of procedure calls on the local state of the caller, in the presence of symbolic constants and pointers. We have implemented polymorphic predicate abstraction in the C2BP tool. We have used C2BP as part of the SLAM toolkit to validate properties of Windows NT device drivers and to find invariants in several programs [Ball and Rajamani 2001].

APPENDIXES

A. KLEENE'S THREE-VALUED LOGIC

Figure 7 presents Kleene's interpretation for conjunction, disjunction, and negation of three-valued logic. This interpretation is used in the programs of the language presented in Section 3.

B. SOUNDNESS

We represent a *program state* as a pair $\sigma = \langle L, \Omega \rangle$, where L is the label on the statement to be executed next and Ω is a store mapping the locations in scope at L to values. We sometimes extend Ω to expressions in the obvious way. An *initial state* of a program P is a state $\langle L, \Omega \rangle$ such that L labels the first statement of a distinguished "main" procedure and Ω maps all locations in scope to appropriate initial values, as described in Section 3.

We say that $\langle L_1, \Omega_1 \rangle \rightarrow \langle L_2, \Omega_2 \rangle$ if execution of the statement at L_1 with store Ω_1 produces store Ω_2 and moves to the statement labeled L_2. The semantics of the $\rightarrow$ relation is standard. A *trace* of a program P is a sequence $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n$ where the first state of the sequence is an initial state of P and where procedure calls and returns are properly matched: execution of a **return** statement transfers control to the statement after the most recent unmatched procedure call. Therefore, a trace represents a valid prefix of an execution of P. Let $\rightarrow^*$ denote the transitive closure of the $\rightarrow$ relation. We sometimes denote traces using a sequence of states with every pair of successive states related by $\rightarrow^*$, when the implicit intermediate states on the trace are of no importance.

Soundness intuitively means that every execution of a program has a corresponding execution in the associated Boolean program abstraction. The main theorem we prove says that one execution step of a program has a corresponding sequence of execution steps in the associated Boolean program abstraction. First, we define a simulation relation on program states, which captures the notion that b-variables should be conservative abstractions of the expressions they represent.

Definition 1. Let P be a program, E be a set of predicates over symbolic constants and variables in P, and $B = BP(P, E)$ be the Boolean program abstraction computed by C2BP. Let V be the b-variables corresponding to predicates in E and $\mathcal{E}$ be the mapping from V to E. We say that a state $\langle L, \Omega \rangle$ of P *is simulated by* a state $\langle L', \Omega' \rangle$ of B if $L = L'$ and for all b-variables $b \in V$ in scope at L' in B we have that:

$$(\Omega'(b) = \textbf{true} \Rightarrow \Omega(\mathcal{E}(b)) = \textbf{true}) \text{ and } (\Omega'(b) = \textbf{false} \Rightarrow \Omega(\mathcal{E}(b)) = \textbf{false}).$$

Our theorem then says that simulation is preserved by the $\rightarrow$ relation.

THEOREM 1. *Let P be a program, E be a set of predicates over symbolic constants and variables in P, and $B = BP(P, E)$ be the Boolean program abstraction computed by* C2BP *. Let V be the b-variables corresponding to predicates in E and $\mathcal{E}$ be the mapping from V to E. Let $\sigma_1 \rightarrow \cdots \rightarrow \sigma_k \rightarrow \sigma$ be a trace of P and $\sigma'_1 \rightarrow^* \cdots \rightarrow^* \sigma'_k$ be a trace of B. If for all $1 \leq i \leq k$ it is the case that σ_i is simulated by σ'_i, then there exists some σ' such that $\sigma'_1 \rightarrow^* \cdots \rightarrow^* \sigma'_k \rightarrow^* \sigma'$ is a trace of B and σ is simulated by σ'.*

PROOF. Since σ_k is simulated by σ'_k, we have that σ_k has the form $\langle L, \Omega_k \rangle$ and σ'_k has the form $\langle L, \Omega'_k \rangle$. Let s be the statement labeled L in P and s' be the statement labeled L in B. Case analysis of the form of s:

—$s = \textbf{skip}$. Then, $s' = \textbf{skip}$ as well. By the semantics of **skip**, we have $\sigma = \langle L_1, \Omega_k \rangle$ where L_1 is the label of the statement following s. Since $s' = \textbf{skip}$, we have that $\sigma'_k \rightarrow \sigma'$, where $\sigma' = \langle L_1, \Omega'_k \rangle$. Then since σ_k is simulated by σ'_k, it follows that σ is simulated by σ'.

—$s = \textbf{goto } L_1$. Then $s' = \textbf{goto } L_1$ as well. By the semantics of **goto**, we have $\sigma = \langle L_1, \Omega_k \rangle$ and $\sigma' = \langle L_1, \Omega'_k \rangle$, where $\sigma'_k \rightarrow \sigma'$. Since σ_k is simulated by σ'_k, it follows that σ is simulated by σ'.

—s = **branch** $\overline{s_1}\| \cdots \|\overline{s_n}$ **end**. Then s' = **branch** $BP(\overline{s_1}, V, \mathcal{E})\| \cdots \|BP(\overline{s_n}, V, \mathcal{E})$ **end**. By the semantics of **branch** we have $\sigma = \langle L_1, \Omega_k \rangle$, where L_1 is the label of the first statement in one of the branch cases. Then, by the semantics of **branch** and the rules for abstracting statement sequences and labeled statements, we have $\sigma'_k \rightarrow \sigma'$, where $\sigma' = \langle L_1, \Omega'_k \rangle$. Since σ_k is simulated by σ'_k, it follows that σ is simulated by σ'.

—s is an assignment statement: Then $\sigma = \langle L_1, \Omega \rangle$, where L_1 is the label of the statement following s. By the abstraction process we know that s' is a parallel assignment, so there exists $\sigma' = \langle L_1, \Omega' \rangle$ such that $\sigma'_k \rightarrow \sigma'$. Now suppose $\Omega'(b)$ is **true**, for some b-variable b in scope at L_1 in B. To finish this case, we show that $\Omega(\mathcal{E}(b))$ is **true** as well. (A similar proof can be done for the case in which $\Omega'(b)$ is **false**.)

Since $\Omega'(b)$ is **true**, we know by the abstraction of assignments that some monomial in $\mathcal{F}_{V,\mathcal{E}}(WP(s, \mathcal{E}(b)))$ was **true** in Ω'_k. Let that monomial be $c_1 \wedge \cdots \wedge c_m$, so for each $1 \leq r \leq m$, $\Omega'_k(c_r) =$ **true**. Since σ_k is simulated by σ'_k, we have that for each $1 \leq r \leq m$, $\Omega_k(\mathcal{E}(c_r)) =$ **true**. By the definition of $\mathcal{F}_{V,\mathcal{E}}$, we know that $\mathcal{E}(c_1) \wedge \cdots \wedge \mathcal{E}(c_m) \Rightarrow WP(s, \mathcal{E}(b))$, so $\Omega_k(WP(s, \mathcal{E}(b))) =$ **true**. Then, by definition of WP, $\Omega(\mathcal{E}(b))$ is **true** as well.

—s is of the form **assume**(e): Then $\sigma = \langle L_1, \Omega_k \rangle$, where L_1 is the label of the statement following s. By the abstraction process, we know that s' is **assume**$(\neg\mathcal{F}_{V,\mathcal{E}}(\neg e))$. Suppose σ'_k can take an evaluation step to some σ'. Then, by the semantics of **assume**, σ' will have the form $\langle L_1, \Omega'_k \rangle$. Since σ_k is simulated by σ'_k, it follows that σ is simulated by σ'.

Therefore, we just need to prove that σ'_k can take an evaluation step to some σ', which is the case according to the semantics of **assume** if and only if $\Omega'_k(\neg\mathcal{F}_{V,\mathcal{E}}(\neg e))$ is not **false**. Since $\sigma_k \rightarrow \sigma$, we know that $\Omega_k(e)$ is **true**. Therefore, $\Omega_k(\neg e)$ is **false**. By definition, $\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(\neg e)) \Rightarrow \neg e$, so also $\Omega_k(\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(\neg e)))$ is **false**. Therefore, $\Omega_k(\neg\mathcal{E}(\mathcal{F}_{V,\mathcal{E}}(\neg e))) = \Omega_k(\mathcal{E}(\neg\mathcal{F}_{V,\mathcal{E}}(\neg e)))$ is **true**. Then since σ_k is simulated by σ'_k, it cannot be the case that $\Omega'_k(\neg\mathcal{F}_{V,\mathcal{E}}(\neg e))$ is **false**.

—s is a procedure call of the form $v := R(a_1, a_2, \ldots, a_j)$: Then, $\sigma = \langle L_1, \Omega \rangle$, where L_1 is the label of the first statement in procedure R and where Ω maps locations involving global variables to their values in Ω_k, maps locations involving formal parameters to the values of the corresponding locations involving actuals in Ω_k, and maps local variables of R to appropriate initial values. By the abstraction process, s' is a call to R', the version of R in B, preceded by assignment statements that compute the values of the actual parameters to the call. Therefore, there exists some $\sigma' = \langle L_1, \Omega' \rangle$, where $\sigma'_k \rightarrow^* \sigma'$ and Ω' maps global b-variables to their values in Ω'_k, maps formal b-parameters to the values of their actuals in Ω'_k, and maps local b-variables of R' to $*$.

Suppose $\Omega'(b)$ is **true**, for some b in scope. To finish this case, we show that also $\Omega(\mathcal{E}(b))$ is **true**. (A similar proof can be done for the case in which $\Omega'(b)$ is **false**.)

—If b is a global b-variable, then we know that $\Omega'(b) = \Omega'_k(b)$, so $\Omega'_k(b)$ is **true**. Then since Ω_k is simulated by Ω'_k we have that $\Omega_k(\mathcal{E}(b))$ is **true**. Since b is global, we know that $\mathcal{E}(b)$ refers only to locations involving

global variables of P. Since for each such location l we have $\Omega(l) = \Omega_k(l)$, also $\Omega(\mathcal{E}(b)) = \Omega_k(\mathcal{E}(b))$. Therefore $\Omega(\mathcal{E}(b))$ is true.

—If b is a formal b-parameter of R', then b has the value of the associated actual parameter to the call. Then by the process for computing the *prm* actual b-variables, some monomial in $\mathcal{F}_{V,\mathcal{E}}(\mathcal{E}(b)[a_1/f_1, \ldots, a_j/f_j])$ was **true** in Ω'_k, where f_i is the associated formal for actual a_i in the call to R in P. Let that monomial be $c_1 \wedge \cdots \wedge c_m$, so for each $1 \leq r \leq m, \Omega'_k(c_r) =$ **true**. Then since σ_k is simulated by σ'_k we have that for each $1 \leq r \leq m, \Omega_k(\mathcal{E}(c_r)) =$ **true**. By the definition of $\mathcal{F}_{V,\mathcal{E}}$ we have $\mathcal{E}(c_1) \wedge \cdots \wedge \mathcal{E}(c_m) \Rightarrow \mathcal{E}(b)[a_1/f_1, \ldots, a_j/f_j]$, so $\Omega_k(\mathcal{E}(b)[a_1/f_1, \ldots, a_j/f_j])$ is **true**. Since b is a formal b-parameter of R', it may only refer to locations involving formal parameters of R and global variables of P. Therefore by the relationship between Ω_k and Ω we have that $\Omega(\mathcal{E}(b))$ is **true**.

—Otherwise, b is a local variable. But by the definition of Ω' we know that b has the value $*$, contradicting the fact that $\Omega'(b)$ is **true**.

—s is a return statement of the form **return** r. Then $\sigma = \langle L_1, \Omega \rangle$, where L_1 is the label of the first statement after the last unmatched procedure call in P's trace. Statement s' is also a **return** statement. Since σ_i is simulated by σ'_i for $1 \leq i \leq k$, the two traces must agree on the last unmatched procedure call. Then there exists an appropriate $\sigma' = \langle L_1, \Omega' \rangle$ such that $\sigma'_k \rightarrow \sigma'$.

Let $\langle \hat{L}, \hat{\Omega} \rangle$ represent the program state before that last unmatched procedure call in P's trace. Similarly, let $\langle \hat{L}, \hat{\Omega}' \rangle$ represent the program state before computing actual parameters for the last unmatched procedure call in B's trace. Let the call in P have the form $v := R(a_1, a_2, \ldots, a_j)$, and let it appear in procedure Q. Then, the call in procedure Q' of B has the form $ret_1, \ldots, ret_k := R'(prm_1, \ldots, prm_p)$, followed by updates of the b-variables associated with expressions in E_u.

Suppose $\Omega'(b)$ is **true**, for some b in scope. To finish this case, we show that also $\Omega(\mathcal{E}(b))$ is **true**. (A similar proof can be done for the case in which $\Omega'(b)$ is **false**.) There are three cases to consider:

—First, suppose b is a global b-variable. Then, b was in the scope of the call. Further, since each b-variable in E_u is a local variable, we know that b is not in E_u. Also, b is not one of the *ret* variables, so b is not assigned to upon return from the call. Therefore by the semantics of **return** we have that $\Omega'(b) = \Omega'_k(b)$. So we have $\Omega'_k(b)$ is **true**, and since σ_k is simulated by σ'_k also $\Omega_k(\mathcal{E}(b))$ is **true**. We know that v is not a global variable (see the definition of internal form in Section 4), so by the semantics of **return** we have that the values of all locations involving global variables agree in Ω_k and Ω. Since b is a global b-variable, $\mathcal{E}(b)$ may only refer to locations involving global variables, so we have $\Omega_k(\mathcal{E}(b)) = \Omega(\mathcal{E}(b))$, and therefore $\Omega(\mathcal{E}(b))$ is **true**.

—Second, suppose b is a local b-variable or formal b-parameter of V_Q and $\mathcal{E}(b) \notin E_u$. By the semantics of procedure calls, $\hat{\Omega}'$ and Ω' agree on the values of local and formal b-variables in V_Q, so we have that $\hat{\Omega}'(b) = \Omega'(b)$ and $\hat{\Omega}'(b)$ is **true**. Then, since $\langle \hat{L}, \hat{\Omega} \rangle$ is simulated by $\langle \hat{L}, \hat{\Omega}' \rangle$, we have $\hat{\Omega}(\mathcal{E}(b))$ is **true**. Since $\mathcal{E}(b) \notin E_u$, by the definition of E_u we have that the

value of $\mathcal{E}(b)$ cannot change as a result of the call to R, so $\hat{\Omega}(\mathcal{E}(b)) = \Omega(\mathcal{E}(b))$. Therefore, $\Omega(\mathcal{E}(b))$ is **true**.

—Finally, suppose b is a local b-variable or formal b-parameter of V_Q and $\mathcal{E}(b) \in E_u$. For this case, we consider the call to R in P to consist of the two statements

$$v_{ret} := R(a_1, a_2, \ldots, a_j); v := v_{ret},$$

where v_{ret} is a fresh local variable in Q. Additionally, just before the call we add a statement of the form $x_o := x$; for each variable x in scope such that $orig(x) = x_o$, where x_o is fresh. Similarly we add a statement of the form $x_{o,n} := *^n x$ for each access expression $*^n x$ in scope such that $orig(*^n x) = x_{o,n}$, where $x_{o,n}$ is fresh. Clearly, the semantics of the call is preserved by these modifications.

Let $\hat{\mathcal{E}}$ be the mapping from b-variables in $\hat{V} = V_Q \cup T_Q$ as defined in Section 4. Let Ω_r be the store at the point in (the revised) P between the two statements above, and let Ω'_r be the store at the point in B just after the call to R', before the updates to the associated b-variables of expressions in E_u. Our strategy is to show that for all b-variables b_0 in the domain of $\hat{\mathcal{E}}$:

$$(\Omega'_r(b_0) = \textbf{true} \Rightarrow \Omega_r(\hat{\mathcal{E}}(b_0)) = \textbf{true}) \quad \text{and}$$
$$(\Omega'_r(b_0) = \textbf{false} \Rightarrow \Omega_r(\hat{\mathcal{E}}(b_0)) = \textbf{false}).$$

Suppose we can show this is the case. Then, since b (the b-variable we are currently considering in E_u) is subsequently assigned in B to $\texttt{choose}(\mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP(v := v_{ret}, \mathcal{E}(b))), \mathcal{F}_{\hat{V},\hat{\mathcal{E}}}(WP(v := v_{ret}, \neg\mathcal{E}(b))))$ and the current statement in P is $v := v_{ret}$, it follows from the same argument as in the case above for assignment statements that $\Omega(\mathcal{E}(b))$ is **true**.

Therefore, suppose $\Omega'_r(b_0) = \textbf{true}$, for some b_0 in the domain of $\hat{\mathcal{E}}$. (A similar proof can be done for the case in which $\Omega'_r(b_0)$ is **false**.) According to the definition of $\hat{\mathcal{E}}$, there are three cases to consider:

—b_0 is a global b-variable in V_Q, and $\hat{\mathcal{E}}(b_0) = \mathcal{E}_Q(b_0) = \mathcal{E}(b_0)$. This case is proven by an analogous argument as used for global b-variables in the first subcase above for the **return** statement.

—b_0 is a local b-variable in V_Q, and $\hat{\mathcal{E}}(b_0) = orig(\mathcal{E}_Q(b_0))$. By the semantics of procedure calls, $\hat{\Omega}'$ and Ω'_r agree on the values of local and formal b-variables in V_Q, so we have that $\hat{\Omega}'(b_0) = \Omega'_r(b_0)$ and $\hat{\Omega}'(b_0)$ is **true**. Then, since $\langle \hat{L}, \hat{\Omega} \rangle$ is simulated by $\langle \hat{L}, \hat{\Omega}' \rangle$, we have $\hat{\Omega}(\mathcal{E}_Q(b_0))$ is **true**. Then, by the assignments $x_o := x$ and $x_{o,n} := *^n x$ before the call, that means that $orig(\mathcal{E}_Q(b_0))$ is true in $\hat{\Omega}$. By the definition of *orig*, all locations in $\mathcal{E}_Q(b_0)$ that can be affected by the call to R are replaced by fresh local *original* variables. Therefore, no locations in $orig(\mathcal{E}_Q(b_0))$ can be affected by the call to R. Further, none of those locations can alias the *ret* variables, since those are fresh. Therefore, $orig(\mathcal{E}_Q(b_0))$ is true in Ω_r.

—b_0 is a local of the form *ret*, and $\hat{\mathcal{E}}(b_0) = \gamma(\mathcal{E}_{RQ}(b_0))$. We need to prove that $\Omega_r(\gamma(\mathcal{E}_{RQ}(b))$ is **true**. Since b_0 is **true** in Ω'_r and is a returned b-variable from R', we know that b_0's counterpart b'_0, which is returned from R' in its **return** statement, is **true** in Ω'_k also. Therefore, since σ_k is simulated by σ'_k we have that $\Omega_k(\mathcal{E}_R(b'_0))$ is **true**. That is equivalent to saying

that $\Omega_k(\mathcal{E}_{RQ}(b_0))$ is **true**. Therefore, we can conclude that $\Omega_r(\gamma(\mathcal{E}_{RQ}(b_0)))$ is **true** if we can show that for all locations l in the domain of γ, $\Omega_k(l) = \Omega_r(\gamma(l))$. We do a case analysis on l. We have three cases:

—$var(l)$ is a global, so $\gamma(l) = l$. By the semantics of **return** and the fact that the return value is assigned to the local variable v_{ret}, the result follows.

—$var(l)$ is the return variable r, so $\gamma(l) = l[v_{ret}/var(l)]$. By the semantics of **return** and the fact that the return value r is assigned to v_{ret}, the result follows.

—$var(l)$ is a symbolic constant $'*^n f$, so $\gamma(l) = l[orig(*^n a)/var(l)]$, where a is the actual for formal f. By the internal form definition, the symbolic constant $'*^n f$ is a local of R' that is assigned the value of $*^n f$ at the entry point of the procedure and never modified. By the definition of *orig*, the value of $orig(*^n a)$ must not be modified by the call, so $\hat{\Omega}(orig(*^n a)) = \Omega_r(orig(*^n a))$. By the definition of *orig* and the assignments to original variables at the call site, we know that $*^n f$ at the entry point of the procedure has the same value as $orig(*^n a)$ before the call, so we have that $\Omega_k('*^n f) = \hat{\Omega}(orig(*^n a))$. So we have shown that $\Omega_k('*^n f) = \Omega_r(orig(*^n a))$, which implies by the semantics of **return** that $\Omega_k(l) = \Omega_r(\gamma(l))$. □

Finally, soundness follows as a simple corollary to the above theorem:

COROLLARY 1 (SOUNDNESS). *Let P be a program, E be a set of predicates over symbolic constants and variables in P, and $B = BP(P, E)$ be the Boolean program abstraction computed by* C2BP. *Let V be the b-variables corresponding to predicates in E and $\mathcal{E}$ be the mapping from V to E. If $\sigma_1 \to \cdots \to \sigma_k$ is a trace of P, then there exists a trace $\sigma'_1 \to^* \cdots \to^* \sigma'_k$ of B such that for all $1 \le i \le k$, σ_i is simulated by σ'_i.*

PROOF. The proof is by induction on k. For the base case, suppose $k = 1$, so the trace in P consists solely of σ_1, an initial state. Let $\sigma_1 = \langle L, \Omega \rangle$, where L is the label of the first statement in the "main" procedure. Then there is an initial state σ'_1 of B, of the form $\langle L, \Omega' \rangle$. Further, since the initial values of b-variables is $*$, all b-variables in the domain of Ω' have value $*$. Therefore, σ_1 is simulated by σ'_1 vacuously.

For the inductive case, suppose k is some $j > 1$, and assume the corollary holds for traces of P of length smaller than j. Since we're given that $\sigma_1 \to \cdots \to \sigma_j$ is a trace of P, so is $\sigma_1 \to \cdots \to \sigma_{j-1}$. Then, by the inductive hypothesis there exists a trace $\sigma'_1 \to^* \cdots \to^* \sigma'_{j-1}$ of B such that for all $1 \le i < j$, σ_i is simulated by σ'_i. Then, by Theorem 1, there exists some σ'_j such that $\sigma'_1 \to^* \cdots \to^* \sigma'_j$ is a trace of B and σ_j is simulated by σ'_j, so the result follows. □

ACKNOWLEDGMENTS

We thank Rupak Majumdar for his hard work in helping the C2BP tool come to life. Andreas Podelski provided a crucial insight that a symbolic constant can be thought of as a local variable to which the corresponding formal is assigned on entry to the procedure.

REFERENCES

BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. K. 2001a. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*. ACM, New York, pp. 203–213.

BALL, T., PODELSKI, A., AND RAJAMANI, S. K. 2001b. Boolean and Cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 2031. Springer-Verlag, New York, pp. 268–283.

BALL, T. AND RAJAMANI, S. K. 2000. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*. Lecture Notes in Computer Science, vol. 1885. Springer-Verlag, New York, pp. 113–130.

BALL, T. AND RAJAMANI, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*. Lecture Notes in Computer Science, vol. 2057. Springer-Verlag, New York, pp. 103–122.

BALL, T. AND RAJAMANI, S. K. 2002a. Generating abstract explanations of spurious counterexamples in C programs. Tech. Rep. MSR-TR-2002-09. Microsoft Research, Jan.

BALL, T. AND RAJAMANI, S. K. 2002b. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*. ACM, New York, pp. 1–3.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *Comput. Surv. 17*, 4, 471–522.

CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2000. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*. Lecture Notes in Computer Science, vol. 1855. Springer-Verlag, New York, pp. 154–169.

CORBETT, J., DWYER, M., HATCLIFF, J., PASAREANU, C., ROBBY, LAUBACH, S., AND ZHENG, H. 2000. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000: International Conference on Software Engineering*. ACM, New York, pp. 439–448.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*. ACM, New York, pp. 238–252.

DAS, M. 2000. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*. ACM, New York, pp. 35–46.

DAS, S., DILL, D. L., AND PARK, S. 1999. Experience with predicate abstraction. In *CAV 00: Computer-Aided Verification*. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, New York, pp. 160–171.

DETLEFS, D., NELSON, G., AND SAXE, J. B. 2003. Simplify: A theorem prover for program checking. HP Labs Technical Report HPL-2003-148.

DIJKSTRA, E. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J.

DWYER, M., HATCLIFF, J., JOEHANES, R., LAUBACH, S., PASAREANU, C., ROBBY, VISSER, W., AND ZHENG, H. 2001. Tool-supported program abstraction for finite-state verification. In *ICSE 01: International Conference on Software Engineering*. ACM, New York, pp. 177–187.

ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. ACM, New York, pp. 234–245.

FLANAGAN, C. AND QADEER, S. 2002. Predicate abstraction for software verification. In *POPL '02: Principles of Programming Languages*. ACM, New York, 191–202.

FOSTER, J. S., FAHNDRICH, M., AND AIKEN, A. 2000. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *SAS 00: Static Analysis*. Lecture Notes in Computer Science, vol. 1824. Springer-Verlag, New York, pp. 175–198.

GRAF, S. AND SAÏDI, H. 1997. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*. Lecture Notes in Computer Science, vol. 1254. Springer-Verlag, New York, pp. 72–83.

GRIES, D. 1981. *The Science of Programming*. Springer-Verlag, New York.
HAVELUND, K. AND PRESSBURGER, T. 1998. Model checking Java programs using Java PathFinder. *Int. J. Softw. Tools Tech. Trans. 2*, 4 (Apr.).
HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND MCMILLAN, K. L. 2004. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, pp. 232–244.
HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *POPL '02: Principles of Programming Languages* (Jan.). ACM, New York, pp. 58–70.
HOLZMANN, G. 1997. The Spin model checker. *IEEE Trans. Softw. Eng. 23*, 5 (May), 279–295.
HOLZMANN, G. 2000. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*. Lecture Notes in Computer Science, vol. 1885. Springer-Verlag, New York, pp. 131–147.
LANDI, W. AND RYDER, B. G. 1992. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices 27*, 7, 235–248.
MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.
MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Mass.
MORRIS, J. M. 1982. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*. Lecture Notes of an International Summer School, Reidel, pp. 25–34.
NELSON, G. 1981. Techniques for program verification. Tech. Rep. CSL81-10. Xerox Palo Alto Research Center, Palo Alto, Calif.
REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*. ACM, New York, pp. 49–61.
SAÏDI, H. AND SHANKAR, N. 1999. Abstract and model check while you prove. In *CAV 99: Computer-aided Verification*. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, New York, pp. 443–454.
SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, New York, pp. 189–233.
VISSER, W., PARK, S., AND PENIX, J. 2000. Using predicate abstraction to reduce object-oriented programs for model checking. In *FMSP 00: Formal Methods in Software Practice*. ACM, New York, pp. 3–12.
WILSON, R. P. AND LAM, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation*. ACM, New York, pp. 1–12.

Received June 2002; revised June 2003; accepted November 2003