Automatic Predicate Abstraction of C Programs

Thomas Ball
tball@microsoft.com
Microsoft Research

Rupak Majumdar
rupak@cs.berkeley.edu
U.C. Berkeley

Todd Millstein
todd@cs.washington.edu
Univ. of Washington

Sriram K. Rajamani
sriram@microsoft.com
Microsoft Research

http://research.microsoft.com/slam/

Abstract

Model checking has been widely successful in validating and debugging designs in the hardware and protocol domains. However, state-space explosion limits the applicability of model checking tools, so model checkers typically operate on abstractions of systems.

Recently, there has been significant interest in applying model checking to software. For infinite-state systems like software, abstraction is even more critical. Techniques for abstracting software are a prerequisite to making software model checking a reality.

We present the first algorithm to automatically construct a *predicate abstraction* of programs written in an industrial programming language such as C, and its implementation in a tool — C2BP. The C2BP tool is part of the SLAM toolkit, which uses a combination of predicate abstraction, model checking, symbolic reasoning, and iterative refinement to statically check temporal safety properties of programs.

Predicate abstraction of software has many applications, including detecting program errors, synthesizing program invariants, and improving the precision of program analyses through predicate sensitivity. We discuss our experience applying the C2BP predicate abstraction tool to a variety of problems, ranging from checking that list-manipulating code preserves heap invariants to finding errors in Windows NT device drivers.

1 Introduction

In the hardware and protocol domains, *model checking* has been used to validate and debug systems by algorithmic exploration of their state spaces. State-space explosion is a major limitation, and typically model checkers explore the state space of an *abstracted* system. For software, which is typically infinite-state, abstraction is even more critical. Any effort to model check software must first construct an abstract model of the software.

A promising approach to construct abstractions automatically, called *predicate abstraction*, was first proposed by Graf and Saïdi [19]. With predicate abstraction, the concrete states of a system are mapped to abstract states according to their evaluation under a finite set of predicates. Automatic predicate abstraction algorithms have been designed and implemented before for finite-state systems and for infinite-state systems specified as guarded commands. However, no one has demonstrated automatic predicate abstraction on a programming language such as C.

We present a tool called C2BP that performs automatic predicate abstraction of C programs. Given a C program P and a set E of predicates (pure C boolean expressions containing no function calls), C2BP automatically creates a *boolean program* $\mathcal{BP}(P,E)$, which is an abstraction of P. A boolean program is essentially a C program in which the only type available is boolean (the boolean program language has some additional constructs that will be presented later). The boolean program has the same control-flow structure as P but contains only $|E|$ boolean variables, each representing a predicate in E. For example, if the predicate $(x < y)$ is in E, where x and y are integer variables in P, then there is a boolean variable in $\mathcal{BP}(P,E)$ whose truth at a program point p implies that $(x < y)$ is true at p in P. For each statement s of P, C2BP automatically constructs the corresponding boolean transfer functions that conservatively represent the effect of s on the predicates in E. The resulting boolean program can be analyzed precisely using a tool called BEBOP [5] that performs interprocedural dataflow analysis [31, 28] using binary decision diagrams.

We present the details of the C2BP algorithm, as well as results from applying C2BP to a variety of problems and programs:

- We have applied C2BP and BEBOP to pointer-manipulating programs to identify invariants involving pointers. In one example, these invariants lead to more precise aliasing information than is possible with a flow-sensitive alias analysis. In another example, we show that list-manipulating code preserves various structural properties of the heap, as has been done with shape analysis [30]. This is noteworthy because our predicate

PLDI 2001, Snowbird, Utah, USA.

language is a quantifier-free logic, rather than the more powerful logic of [30].

- We have applied C2BP and BEBOP to examples from Necula's work on proof-carrying code [26] to automatically identify loop invariants in these examples that the PCC compiler was required to generate.
- We have used C2BP in the SLAM toolkit to check temporal safety properties of Windows NT device drivers. The SLAM toolkit uses C2BP and BEBOP to statically determine whether or not an assertion violation can take place in C code. A unique part of the toolkit is its use of a demand-driven iterative process to automatically find predicates that are relevant to the particular assertion under examination. When the current set of predicates and the boolean program abstraction that it induces are insufficient to show that an assertion does/doesn't fail, new predicates are found to refine the abstraction. Although the SLAM process may not converge in theory, due to the undecidability of the assertion violation problem, it has converged on all NT device drivers we have analyzed (even though they contain loops).

For a detailed proof of soundness of the abstraction algorithm presented in this paper, the interested reader is referred to our technical report [3]. In work with Andreas Podelski [4] we have used the framework of abstraction interpretation to formalize the precision of the C2BP algorithm for single procedure programs with no pointers. Section 4.6 reviews the soundness theorem for C2BP that we have proved and describes our precision results.

The rest of this paper is organized as follows. Section 2 gives an example of applying C2BP to a pointer-manipulating C procedure. Section 3 lists the challenges in performing predicate abstraction on C programs. Section 4 describes our predicate abstraction algorithm in detail. Section 5 describes extensions and optimizations to the C2BP tool. Section 6 presents results on applying the C2BP tool to a variety of C programs. Section 7 reviews related work and Section 8 concludes the paper.

2 Example: Invariant Detection in Pointer-manipulating Programs

This section presents the application of C2BP and the BEBOP model checker to a pointer-manipulating procedure. The combination of the two tools determines program-point-specific invariants about the procedure, which can be used to refine pointer aliasing information.

2.1 C2bp

Consider the `partition` function of Figure 1(a). This procedure takes a pointer to a list of integers l and an integer v and partitions the list into two lists: one containing the cells with value greater than v (returned by the function) and the other containing the cells with value less than or equal to v (the original list, destructively updated).

We input the program in Figure 1(a) along with the following predicate input file to C2BP:

```
partition {
  curr == NULL,
  prev == NULL,
  curr->val > v,
  prev->val > v
}
```

The predicate input file specifies a set of four predicates, local to the procedure `partition`. Figure 1(b) shows the boolean program resulting from the abstraction of the procedure `partition` with respect to these predicates.[1] The boolean program declares four variables of type `bool` in procedure `partition`, each corresponding to one of the four predicates from the predicate input file.[2] The variables' initial values are unconstrained.

The boolean program is guaranteed to be an abstraction of the C program in the following sense: any feasible execution path of the C program is a feasible execution path of the boolean program. Of course, there may be feasible execution paths of the boolean program that are infeasible in the C program. Such paths can lead to imprecision in subsequent model checking.

We now informally describe how the C2BP tool translates each statement of the C program into a corresponding set of statements in the boolean program. An assignment statement in the C program is translated to a set of assignments that capture the effect of the original assignment statement on the input predicates. For example, the assignment statement "`prev=NULL;`" in the C program is translated to two assignment statements in the boolean program. The first, "`{prev==NULL}=true;`", reflects the truth of the predicate ($prev$ = **NULL**) after the assignment. The value of the predicate ($prev \rightarrow val > v$) is undefined after this assignment and is thus invalidated by the assignment statement "`{prev->val>v} = unknown();`". The `unknown` function is defined as:

```
bool unknown() {
  if (*) { return true; }
  else { return false; }
}
```

The `unknown` function uses the control expression "*", which non-deterministically chooses the `then` or the `else` branch, to return either **true** or **false**.

The C2BP tool determines that the other two predicates are unaffected by the assignment "`prev=NULL;`", so they need not be updated. The C2BP tool uses a flow-insensitive points-to analysis [12] to resolve aliases between pointers. In this program, since none of the pointer variables in the set { `curr`, `prev`, `next`, `newl` } has its address taken, none of these variables can be aliased by any other expression in the procedure. As a result, C2BP resolves that the only predicates that the assignment "`prev=NULL;`" affects are ($prev$ = **NULL**) and ($prev \rightarrow val > v$).

As another example, the assignment "`prev=curr;`" is also abstracted to assignments to the two predicates involving `prev`. These predicates are assigned the values of the corresponding predicates on `curr`, as expected. Finally, C2BP determines that the assignment "`newl=NULL;`" cannot affect *any* of the four input predicates, so the assignment is translated to the `skip` statement, the boolean program's "no-op".

In the above examples, most of the input predicates are updated accurately. For example, the assignment

[1] The boolean program shown is not the exact output of C2BP— it has been simplified to aid readability.

[2] In boolean programs, variable identifiers can be regular C identifiers or an arbitrary string enclosed between "{" and "}".

```
 typedef struct cell {
   int val;
   struct cell* next;
 } *list;

 list partition(list *l, int v) {
   list curr, prev, newl, nextCurr;

   curr = *l;
   prev = NULL;
   newl = NULL;
   while (curr != NULL) {
     nextCurr = curr->next;
     if (curr->val > v) {
       if (prev != NULL) {
         prev->next = nextCurr;
       }
       if (curr == *l) {
         *l = nextCurr;
       }
       curr->next = newl;
L:     newl = curr;
     } else {
       prev = curr;
     }
     curr = nextCurr;
   }
   return newl;
 }
```

(a)

```
 void partition() {
   bool {curr==NULL},  {prev==NULL};
   bool {curr->val>v}, {prev->val>v};
   {curr==NULL}  = unknown();           // curr = *l;
   {curr->val>v} = unknown();
   {prev==NULL} = true;                 // prev = NULL;
   {prev->val>v} = unknown();
   skip;                                // newl = NULL;
   while(*) {                           // while(curr!=NULL)
     assume(!{curr==NULL});             //
     skip;                              //  nextCurr = curr->next
     if (*) {                           //  if (curr->val > v) {
       assume({curr->val>v});           //
       if (*) {                         //   if (prev != NULL) {
         assume(!{prev==NULL});         //
         skip;                          //    prev->next = nextCurr;
       }                                //   }
       if (*) {                         //   if (curr == *l) {
         skip;                          //    *l = nextCurr;
       }                                //   }
       skip;                            //   curr->next = newl;
L:     skip;                            //   newl = curr
     } else {                           //  } else {
       assume(!{curr->val>v});          //
       {prev==NULL} = {curr==NULL};     //  prev = curr;
       {prev->val>v} = {curr->val>v};   //
     }                                  //  }
     {curr==NULL} = unknown();          //  curr = nextCurr;
     {curr->val>v} = unknown();
   }
   assume({curr==NULL});
 }
```

(b)

Figure 1: (a) List partition example; (b) The boolean program of the list partition example, abstracted with respect to the set of input predicates { `curr==NULL, prev==NULL, curr->val > v, prev->val > v` }. The `unknown` function is used to generate the value **true** or **false** non-deterministically (see body text for an explanation).

"`{prev==NULL}={curr==NULL};`" in the boolean program exactly represents the effect of the assignment "`prev=curr`" on the predicate ($prev = \mathbf{NULL}$). However, it is possible for such exact information to be unavailable, because some of the necessary predicates have not been input to C2BP. In that case, we must replace exact information with a conservative approximation. For example, the assignment "`curr=nextCurr;`" can affect the two predicates involving `curr`. However, because there are no predicates about `nextCurr` in the predicate input file, there is no way to deduce the correct truth value of these predicates. This represents a worst case of sorts, as the input predicates provide absolutely no information about the appropriate truth values for the two predicates to be updated. As a result, the two predicates are "invalidated" using the `unknown` function, as defined above.

The C2BP tool translates conditional statements in the C program into non-deterministic conditional statements in the boolean program, using the control expression "*". However, it also inserts "assume" statements to capture the semantics of conditionals with respect to the input predicates. For example, the first statement inside the `while` loop is "`assume(!{curr==NULL});`". The `assume` acts as a filter on the state space of the boolean program: in this case, it is impossible to reach the program point after the `assume` if the variable {`curr==NULL`} is true. In this way, we faithfully model the guard of the original `while` loop.

2.2 Bebop

The boolean program output by C2BP is input to the Bebop model checker [5], which computes the set of reachable states for each statement of a boolean program using an interprocedural dataflow analysis algorithm in the spirit of Sharir-Pnueli and Reps-Horwitz-Sagiv [31, 28]. A state of a boolean program at a statement s is simply a valuation to the boolean variables that are in scope at statement s (in other words, a bit vector, with one bit for each variable in scope). The set of reachable states (or invariant) of a boolean program at s is thus a set of bit vectors (equivalently, a boolean function over the set of variables in scope at s).

Bebop differs from typical implementations of dataflow algorithms in two crucial ways. First, it computes over sets of bit vectors at each statement rather than single bit vectors. This is necessary to capture correlations between variables. Second, it uses binary decision diagrams [9] (BDDs) to implicitly represent the set of reachable states of a program, as well as the transfer functions for each statement in a boolean program. However, Bebop uses an explicit control-flow graph representation, as in a compiler, rather than encoding the control-flow with BDDs, as done in most symbolic model checkers.

For our example, Bebop outputs the following invariant representing the reachable states at label `L` of the boolean program:

$$(curr \neq \textbf{NULL}) \ \wedge \ (curr \rightarrow val > v) \ \wedge$$
$$((prev \rightarrow val \leq v) \ \vee \ (prev = \textbf{NULL}))$$

Because C2BP is sound, this boolean function is also an invariant over the state of the C program at label `L`.

Such invariants can be used for many different purposes; we give several examples in Section 6. One interesting usage of the above invariant is to refine alias information. In particular, the invariant implies that `*prev` and `*curr` are never aliases at label `L` in the procedure `partition`. In other words, variables `prev` and `curr` never point to the same memory location at label `L`. This can be seen as follows:

- If $(prev = \textbf{NULL})$, then $(prev \neq curr)$ because $(curr \neq \textbf{NULL})$.
- If $(prev \neq \textbf{NULL})$, then since $(curr \rightarrow val > v)$ and $(prev \rightarrow val \leq v)$, it follows that $(prev \rightarrow val \neq curr \rightarrow val)$, which implies $(prev \neq curr)$.[3]

This fact can be deduced automatically from the given invariant. In particular, a decision procedure can determine that the invariant implies $(prev \neq curr)$. In this way, we can automatically refine an existing alias analysis. Traditional flow-sensitive alias analyses would not discover that `*prev` and `*curr` are not aliases at label `L`, since such analyses do not use the values of fields (such as `prev->val`) to eliminate possible aliasing relationships.

2.3 Summary

We have shown how C2BP is used to compute a boolean program that is a sound abstraction of a C program with respect to a set of predicates E. Subsequent model checking of the boolean program can discover strong invariants that are expressed as boolean functions over the predicates in E.

3 The Challenges of Predicate Abstraction for C

The complexities of a programming language like C gives rise to several technical challenges in performing predicate abstraction:

- **Pointers.** There are two closely related subproblems in dealing with pointers: (1) assignments through dereferenced pointers in the original C program, and (2) pointers and pointer dereferences in the predicates over which the abstraction is computed. We handle the two cases in a uniform manner and describe how to use points-to analysis [12] to improve the precision of our abstraction.
- **Procedures.** Programs with procedures are handled by allowing procedural abstraction in the target language [5]. In particular, boolean programs have global variables, procedures with local variables, and call-by-value parameter passing. Having explicit procedures allows us to make both abstraction and analysis more efficient by exploiting procedural abstraction present in the C program. It also allows us to handle recursive and mutually recursive procedures with no additional mechanism. This differs from most other approaches to software model checking, which inline procedure calls [10]. In the following section, we describe a modular abstraction process for procedures: each procedure can be abstracted given only the *signatures* of the abstractions of its callees, and such signatures can be constructed for each procedure in isolation.
- **Procedure calls.** The abstraction process for procedure calls is challenging, particularly in the presence of pointers. After a call, the caller must conservatively update local state that may have been modified by the callee. We provide a sound and precise approach to abstracting procedure calls that takes such side-effects into account.
- **Unknown values.** It is not always possible to determine the effect of a statement in the C program on a predicate, in terms of the input predicate set E. We deal with such non-determinism directly in the boolean program via the non-deterministic control expression "*", which allows us to implicitly express a three-valued domain for boolean variables.
- **Precision-efficiency tradeoff.** Computing the abstract transfer function for each statement in the C program with respect to the set E of predicates may require the use of a theorem prover. Obtaining a precise abstract transfer function requires $O(2^{|E|})$ calls to the theorem prover, in the worst case. We have explored several optimization techniques to reduce the number of calls made to the theorem prover. Some of these techniques result in an equivalent boolean program, while others trade off precision for computation speed.

4 Predicate Abstraction

This section describes the design and implementation of C2BP in detail. Given a C program P and a set $E = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ of pure boolean C expressions over the variables of P and constants of the C language, C2BP automatically constructs an abstraction of P with respect to E [19]. This abstraction is represented as a *boolean program* $\mathcal{BP}(P,E)$, which is a program that has identical control structure to P but contains only boolean variables. In particular, $\mathcal{BP}(P,E)$ contains n boolean variables $V = \{b_1, b_2, \ldots, b_n\}$, where each boolean variable b_i represents the predicate φ_i $(1 \leq i \leq n)$. As described in Section 4.6, $\mathcal{BP}(P,E)$ is guaranteed to be an abstraction of P in that the set of execution traces of $\mathcal{BP}(P,E)$ is a superset of the set of execution traces of P.

Our tool handles all syntactic constructs of the C language, including pointers, structures, and procedures. Its main limitation is that it uses a *logical* model of memory when analyzing C programs. That is, it models the expression $p+i$, where p is a pointer and i is an integer, as yielding a pointer value that points to the object pointed to by p.

In the sequel, we assume that the C program has been converted into a simple intermediate form in which: (1) all intraprocedural control-flow is accomplished with **if-then-else** statements and **gotos**; (2) all expressions are free of side-effects and short-circuit evaluation and do not contain multiple dereferences of a pointer (e.g., `**p`); (3) a function

[3] Here we use the contrapositive of the rule usually applied in unification-based alias analysis: $(p = q) \Rightarrow (*p = *q)$. That is, $(*p \neq *q) \Rightarrow (p \neq q)$.

call only occurs at the top-most level of an expression (for example, "`z=x+f(y);`" is replaced by "`t=f(y); z=x+t;`").

4.1 Weakest Preconditions and Cubes

For a statement s and a predicate φ, let $WP(s,\varphi)$ denote the *weakest liberal precondition* [16, 20] 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 after s terminates (if it terminates). Let "`x = e`" be an assignment, where x is a scalar variable and e is an expression of the appropriate type. Let φ be a predicate. By definition $WP(\texttt{x = e},\varphi)$ is φ with all occurrences of x replaced with e, denoted $\varphi[e/x]$. For example:

$$WP(\texttt{x=x+1}, x<5) \;=\; (x+1)<5 \;=\; (x<4)$$

The weakest precondition computation is central to the predicate abstraction process. Suppose statement s occurs between program points p and p'. If φ is a predicate in E with corresponding boolean variable b then it is safe to assign b the value **true** in $\mathcal{BP}(P,E)$ between program points p and p' if the boolean variable b' corresponding to $WP(s,\varphi)$ is **true** at program point p. However, no such variable b' may exist if $WP(s,\varphi)$ 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. In this case, C2BP uses 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** afterwards.

We formalize this strengthening of a predicate as follows. A *cube* over V is a conjunction $c_{i_1}\wedge\ldots\wedge c_{i_k}$, where each $c_{i_j}\in\{b_{i_j},\neg b_{i_j}\}$ for some $b_{i_j}\in V$. For a variable $b_i\in V$, let $\mathcal{E}(b_i)$ denote the corresponding predicate φ_i, and let $\mathcal{E}(\neg b_i)$ denote the predicate $\neg\varphi_i$. Extend $\mathcal{E}$ to cubes and disjunctions of cubes in the natural way. For any predicate φ and set of boolean variables V, let $\mathcal{F}_V(\varphi)$ denote the largest disjunction of cubes c over V such that $\mathcal{E}(c)$ implies φ. The predicate $\mathcal{E}(\mathcal{F}_V(\varphi))$ represents the weakest predicate over $\mathcal{E}(V)$ that implies φ. In our example, $\mathcal{E}(\mathcal{F}_V(x<4))=(x=2)$.

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

For each cube, the implication check involves a call to a theorem prover implementing the required decision procedures. Our implementation of C2BP uses two theorem provers: Simplify [15] and Vampyre [7], both Nelson-Oppen style provers [27]. A naive computation of $\mathcal{F}_V(\cdot)$ and $\mathcal{G}_V(\cdot)$ requires exponentially many calls to the theorem prover in the worst case. Section 5 describes several optimizations that make the $\mathcal{F}_V$ and $\mathcal{G}_V$ computations practical.

4.2 Pointers and aliasing

In the presence of pointers, $WP(\texttt{x=e},\varphi)$ is not necessarily $\varphi[e/x]$. As an example, $WP(\texttt{x = 3}, *p>5)$ is not $(*p>5)$ because if x and $*p$ are aliases, then $(*p>5)$ cannot be true after the assignment to x. A similar problem occurs when a pointer dereference is on the left-hand side of the assignment.

To handle these problems, we adapt Morris' general axiom of assignment [25]. A *location* is either a variable, a structure field access from a location, or a dereference of a location. Consider the computation of $WP(\texttt{x=e},\varphi)$, where x is a location, and let y be a location mentioned in the predicate φ. Then there are two cases to consider: either x and y are aliases, and hence the assignment of e to x will cause the value of y to become e; or they are not aliases, and the assignment to x leaves y unchanged. Define

$$\varphi[x,e,y] = \begin{array}{l}(\;\&x=\&y\;\wedge\;\varphi[e/y])\vee\\(\;\&x\neq\&y\;\wedge\;\varphi)\end{array}$$

Let $y_1,y_2,\ldots,y_n$ be the locations mentioned in φ. Then $WP(\texttt{x=e},\varphi)$ is defined to be $\varphi[x,e,y_1][x,e,y_2]\ldots[x,e,y_n]$. In the example above, we have

$$\begin{array}{c}WP(\texttt{x = 3}, *p>5)=\\(\;\&x=p\wedge 3>5)\vee(\;\&x\neq p\wedge *p>5)\end{array}$$

In the absence of alias information, if the predicate φ has k locations occurring in it, the weakest precondition will have 2^k syntactic disjuncts, each disjunct considering a possible alias scenario of the k locations with x. C2BP uses a pointer analysis to improve the precision of the weakest precondition computation. If the pointer analysis says that x and y cannot be aliased at the program point before `x=e`, then we can prune the disjuncts representing a scenario where x is aliased to y, and we can partially evaluate the disjuncts representing a scenario where x is not aliased to y. This has the effect of improving the precision of the resulting boolean program $\mathcal{BP}(P,E)$ produced by C2BP. Our implementation uses Das's points-to algorithm [12] to obtain flow-insensitive, context-insensitive may-alias information.

4.3 Predicate Abstraction of Assignments

Consider an assignment statement "`x = e;`" at label ℓ in P. The boolean program $\mathcal{BP}(P,E)$ produced by C2BP will contain at label ℓ a parallel assignment to the boolean variables in scope at ℓ. A boolean variable b_i in $\mathcal{BP}(P,E)$ can have the value **true** after ℓ if $\mathcal{F}_V(WP(\texttt{x = e},\varphi_i))$ holds before ℓ. Similarly, b_i can have the value **false** after ℓ if $\mathcal{F}_V(WP(\texttt{x = e},\neg\varphi_i))$ holds before ℓ. Note that these two predicates cannot be simultaneously true. Finally, if neither of these predicates holds before ℓ, then b_i should be set

<pre>int bar(int* q, int y) {
  int l1, l2;
  ...
  return l1;
}</pre>
	<pre>bar {
	  y >= 0,
	  *q <= y,
	  y == l1,
	  y > l2
	}</pre>

<pre>void foo(int* p, int x) {
  int r;
  if (*p <= x)
    *p = x;
  else
    *p = *p + x;
  r = bar(p, x);
  ...
}</pre>
	<pre>foo {
	 *p <= 0,
	  x == 0,
	  r == 0
	}</pre>

Figure 2: An example input to C2BP. On the left are two simple C procedures (`bar` is not shown in its entirety). On the right is the set of predicates to model.

non-deterministically. This can happen because the predicates in E are not strong enough to provide the appropriate information, or because the theorem prover is incomplete. Therefore, $\mathcal{BP}(P, E)$ contains the following parallel assignment at label ℓ:

$$\begin{array}{l} b_1, \ldots, b_n = \\ \quad \texttt{choose}(\mathcal{F}_V(WP(\texttt{x=e}, \varphi_1)), \mathcal{F}_V(WP(\texttt{x=e}, \neg\varphi_1))), \\ \quad \ldots, \\ \quad \texttt{choose}(\mathcal{F}_V(WP(\texttt{x=e}, \varphi_n)), \mathcal{F}_V(WP(\texttt{x=e}, \neg\varphi_n))) \end{array}$$

where the `choose` function is always part of $\mathcal{BP}(P, E)$ and is defined as follows:

```
bool choose(bool pos, bool neg) {
  if (pos) { return true; }
  if (neg) { return false; }
  return unknown();
}
```

For example, consider abstracting the statement "`*p=*p+x`" in procedure `foo` of Figure 2 with respect to the three predicates declared to be local to `foo`. Let us call this statement s. In this example, a may-alias analysis reveals that $*p$ cannot alias x or r. The weakest precondition $WP(s, *p \leq 0)$ is $(*p + x) \leq 0$, since `*p` cannot alias `x`. We have $\mathcal{E}(\mathcal{F}_V(*p + x \leq 0)) = (*p \leq 0) \wedge (x = 0)$. Similarly, $WP(s, \neg(*p \leq 0))$ is $\neg((*p + x) \leq 0)$, and $\mathcal{E}(\mathcal{F}_V(\neg(*p + x \leq 0))) = \neg(*p \leq 0) \wedge (x = 0)$. The weakest preconditions of s with respect to the predicates $(x = 0)$ and $(r = 0)$ are the respective predicates themselves, since $*p$ cannot alias x or r. Thus, $\mathcal{BP}(P, E)$ will contain the following statement in place of the given assignment statement, where we use $\{e\}$ to denote the boolean variable representing predicate e:

```
{*p<=0}, {x==0}, {r==0} =
   choose({*p<=0} && {x==0}, !{*p<=0} && {x==0}),
   choose({x==0}            , !{x==0}),
   choose({r==0}            , !{r==0});
```

Note that the abstraction process for assignment statements is based on weakest precondition computations that are *local* to each assignment and can be computed by a purely syntactic manipulation of predicates. C2BP does not compute compositions of weakest preconditions over paths with complex control flow. In particular, C2BP does not require programs to be annotated with function *pre*- or *post*-conditions, or with loop invariants.

4.4 Predicate Abstraction of Gotos and Conditionals

Every **goto** statement in the C program is simply copied to the boolean program.

Translating conditionals is more involved. Consider some conditional `if` (φ) $\{\ldots\}$ `else` $\{\ldots\}$ in program P. At the beginning of the *then* branch in P, the predicate φ holds. Therefore, at the beginning of the *then* branch in the corresponding conditional in $\mathcal{BP}(P, E)$, the condition $\mathcal{G}_V(\varphi)$ is known to hold. Similarly, at the beginning of the *else* branch in P, we know that $\neg\varphi$ holds, so $\mathcal{G}_V(\neg\varphi)$ is known to hold at that program point in $\mathcal{BP}(P, E)$. Therefore, $\mathcal{BP}(P, E)$ will contain the following abstraction of the above conditional:

```
if(*){
   assume(G_V(φ))
   ...
}else{
   assume(G_V(¬φ))
   ...
}
```

Note that the test in the abstracted conditional is $*$, so both paths through the conditional are possible. Within the *then* and *else* branches, we use the `assume` statement to retain the semantics of the original conditional test. The `assume` statement is the dual of `assert`: `assume`(φ) never fails. Executions on which φ does not hold at the point of the `assume` are simply ignored [16].

As an example, consider the conditional in procedure `foo` of Figure 2. The abstraction of this conditional with respect to the three predicates local to `foo` is:

```
if (*) { // if (*p <= x)
    assume ({x == 0}  ==>  {*p <= 0});
    ...
} else {
    assume ({x == 0}  ==>  !{*p <= 0});
    ...
}
```

4.5 Predicate Abstraction of Procedure Calls

We now describe how C2BP handles multi-procedure programs.

4.5.1 Notation

Recall that the input to C2BP is the program P and a set E of predicates. Let G_P be the global variables of the program P. Each predicate in E is annotated as being either global to $\mathcal{BP}(P, E)$ or local to a particular procedure in $\mathcal{BP}(P, E)$ (see Figure 2, in which predicates are local to `bar` or `foo` – there are no global predicates in this example), thereby determining the scope of the corresponding boolean variable in $\mathcal{BP}(P, E)$. A global predicate can refer only to variables in G_P. Let E_G denote the global predicates of E and let V_G denote the corresponding global boolean variables of $\mathcal{BP}(P, E)$.

For a procedure R, let E_R denote the subset of predicates in E that are local to R, and let V_R denote the corresponding local boolean variables of R in $\mathcal{BP}(P, E)$. In the following, we do not distinguish between a boolean variable b and its corresponding predicate $\mathcal{E}(b)$ when unambiguous from the context (that is, in the context of $\mathcal{BP}(P, E)$ we always mean b and in the context of P we always mean $\mathcal{E}(b)$). Let F_R be the formal parameters of R, and let L_R be the local variables of R. Let $r \in L_R \cup F_R$ be the return variable of R (we assume, without loss of generality, that there is only one return statement in R, and it has the form "`return r`").

Let $vars(e)$ be the set of variables referenced in expression e. Let $drfs(e)$ be the set of variables dereferenced in expression e.

4.5.2 Determining signatures

A key feature of our approach is modularity: each procedure can be abstracted by C2BP given only the signatures of procedures that it calls. The signature of procedure R can be determined in isolation from the rest of the program, given E_R. C2BP operates in two passes. In the first pass it determines the signature of each procedure. It uses these signatures to abstract procedure calls (along with all other statements) in the second pass.

Let R be a procedure in P and let R' be its abstraction in $\mathcal{BP}(P, E)$. The *signature* of procedure R is a four-tuple (F_R, r, E_f, E_r), where:

- F_R is the set of formal parameters of R,
- r is the return variable of R,
- E_f is the set of formal parameter predicates of R', defined as $\{e \in E_R \mid \mathit{vars}(e) \cap L_R = \emptyset\}$, and
- E_r is the set of return predicates of R', defined as:

$$\{e \in E_R \mid (r \in \mathit{vars}(e) \wedge (\mathit{vars}(e) \setminus \{r\} \cap L_R = \emptyset)) \vee (e \in E_f \wedge (\mathit{vars}(e) \cap G_P \neq \emptyset \vee \mathit{drfs}(e) \cap F_R \neq \emptyset))\}.$$

E_f is the set of formal parameter predicates of R'. This is the subset of predicates in E_R that do not refer to any local variables of R. All predicates in $E_R - E_f$ will be locals of R'. E_r is the set of predicates to be returned by R' (boolean programs allow procedures to have multiple return values). Such return predicates serve two purposes. One is to provide callers with information about r, the return value of R. The other purpose is to provide callers with information about any global variables and call-by-reference parameters, so that local predicates of callers can be updated precisely. To handle the first concern, E_r contains those predicates in E_R that mention r but do not mention any (other) locals of R in P, as callers will not know about these locals. To handle the second concern, E_r contains those predicates in E_f that reference a global variable or dereference a formal parameter of R.

As an example, consider procedure `bar` in Figure 2. In the signature of `bar`, E_f is $\{*q \leq y, y \geq 0\}$ and E_r is $\{y = l1, *q \leq y\}$.

4.5.3 Handling procedure calls

Consider a call $v = R(a_1, \ldots, a_j)$ to procedure R at label ℓ of some procedure S in P. The abstraction $\mathcal{BP}(P, E)$ contains a call to R' at label ℓ. Let the signature of R be (F_R, r, E_f, E_r). For each formal parameter predicate $e \in E_f$, C2BP computes an actual value to pass into the call. Let

$$e' = e[a_1/f_1, a_2/f_2, \ldots, a_j/f_j]$$

where $F_R = \{f_1, f_2, \ldots, f_j\}$. The expression e' represents the predicate e translated to the calling context. The actual parameter computed for the formal e is

$$\texttt{choose}(\mathcal{F}_{V_S \cup V_G}(e'), \mathcal{F}_{V_S \cup V_G}(\neg e')).$$

We now explain how C2BP handles the return values from the call to R'. Assume $E_r = \{e_1, \ldots, e_p\}$. C2BP creates p fresh local variables $T = \{t_1, \ldots, t_p\}$ in procedure S' and assigns to them, in parallel, the return values of R':

$$\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{p} = \mathtt{R}'(\ldots);$$

The final step is to update each local predicate of S whose value may have changed as a result of the call. Any predicate in E_S that mentions v must be updated. In addition, we must update any predicate in E_S that mentions a global variable, a (possibly transitive) dereference of an actual parameter to the call, or an alias of either of these kinds of locations. C2BP uses the pointer alias analysis to determine a conservative over-approximation E_u to this set of predicates to update.

Let $E' = (E_S \cup E_G) - E_u$. The predicates in E' along with the predicates in E_r are used to update the predicates in E_u. Let $V' \subseteq V_S \cup V_G$ be the boolean variables in $\mathcal{BP}(P, E)$ corresponding to E'.

First C2BP translates the predicates in E_r to the calling context. In particular, for each $e_i \in E_r$, let

$$e'_i = e_i[v/r, a_1/f_1, a_2/f_2, \ldots, a_j/f_j]$$

and let $E'_r = \{e'_1, \ldots, e'_p\}$.[4] Define $\mathcal{E}(t_i) = e'_i$, for each $t_i \in T$. For each $e \in E_u$, the corresponding boolean variable $b \in V_S$ is assigned the following value:

$$\texttt{choose}(\mathcal{F}_{V' \cup T}(e), \mathcal{F}_{V' \cup T}(\neg e)).$$

For example, consider the call "`bar(p,x)`" in Figure 2. Recall that in the signature of `bar`, the formal parameter predicates (E_f) are $\{*q \leq y, y \geq 0\}$ and the return predicates (E_r) are $\{y = l1, *q \leq y\}$. The abstraction of this call in the boolean program is as follows:

```
prm1 = choose({*p<=0}&&{x==0}, // for formal {*q<=y}
              !{*p<=0}&&{x==0});
prm2 = choose({x==0}, false);  // for formal {y>=0}

t1, t2 = bar(prm1, prm2);       // t1 for {*q<=y}
                                // t2 for {y==l1}

{*p<=0} =  choose(t1&&{x==0}, !t1&&{x==0});
{r==0}  =  choose(t2&&{x==0}, !t2&&{x==0};
```

4.6 Formal properties

We give two properties that relate P and $\mathcal{BP}(P, E)$. The first property, soundness, states that B is an abstraction of P —every feasible path in P is feasible in B as well. Since a boolean program that allows all paths to be feasible is sound as well, we also need to state the sense in which B is precise. We do that via the terminology of abstract interpretation [11].

Soundness. For any path p feasible in P, it is guaranteed that p is feasible in $\mathcal{BP}(P, E)$ as well. Further, if Ω is the state of the C program P after executing path p, then there exists an execution of p in the boolean program B ending in a state Γ such that for every $1 \leq i \leq n$, φ_i holds in Ω iff b_i is true in Γ. A proof of the soundness of C2BP can be found in [3].

Precision. The framework of abstract interpretation can be used to specify abstractions declaratively. A *boolean abstraction* maps concrete states to abstract states according to their evaluation under a finite set of predicates. A *cartesian abstraction* maps a set of boolean vectors to a three-valued vector obtained by ignoring dependencies between the components of the vectors (see, for example, the work on set-based analysis [21]). For example, the set of boolean

[4] For simplicity, we assume that each formal still refers to the same value as its corresponding actual at the end of the call. This can be checked using a standard modification side-effect analysis [24]. If a formal cannot be proven to refer to the same value as its corresponding actual at the end of the call, then any predicates that mention the formal must be removed from E_r in the signature of R.

vectors $\{(0,1),(1,0)\}$ is mapped by the cartesian abstraction to the three-valued vector $(?,?)$, where ? represents the "don't know" value. For single procedures without pointers, the abstraction computed by C2BP is equivalent to a composition of the boolean and cartesian abstractions [4]. We improve precision by using disjunctive completion and focus operations, both of which are implemented in BEBOP using BDDs [4].

5 Extensions

This section describes various techniques we have applied to increase the precision and efficiency of C2BP.

5.1 The `enforce` construct

Often the predicates in E are correlated in some way. For example, consider the predicates $(x = 1)$ and $(x = 2)$. The semantics associated with these predicates forbids the predicates from being simultaneously **true**. However, when we use uninterpreted boolean variables b_1 and b_2 for the predicates in $\mathcal{BP}(P, E)$, we do not preclude an execution of the boolean program in which both variables evaluate **true** in some state. In order to rule out abstract executions containing such spurious situations, we add an `enforce` construct to boolean programs: the statement `enforce` θ in a procedure has the effect of putting `assume` θ between every statement in the procedure. This ensures that θ is a data invariant maintained throughout the procedure's execution. We compute θ for each procedure R simply as $\mathcal{F}_{V_R \cup V_G}(\textbf{false})$. For example, given only predicates $(x = 1)$ and $(x = 2)$, $\mathcal{E}(\theta)$ is $\neg((x = 1) \wedge (x = 2))$.

5.2 Optimizations

The method described above for constructing abstract models of C programs is impractical without several important optimizations. Profiling shows that the running time of C2BP is dominated by the cost of theorem proving, as we are making an exponential number of calls to the prover at each program point. Therefore, our optimization efforts have focused on cutting down the number of calls to the theorem prover.

First, when computing $\mathcal{F}_V(\varphi)$, cubes are considered in increasing order by length. If a cube c is shown to imply φ, then we know that any cube that contains c as a subset will also imply φ, is redundant with c, and can therefore be safely pruned. In this way, the $\mathcal{F}$ computation actually produces a disjunction of only the *prime implicants* of $\mathcal{F}_V(\varphi)$. If a cube c does not imply φ but it implies $\neg\varphi$, then any cube that contains c as a subset also will not imply φ, and can therefore be safely pruned.

Second, for every assignment statement, rather than updating the values of every boolean variable in scope, we do not update those variables whose truth value will definitely not change as a result of the assignment. The truth value of a variable b will definitely not change as a result of an assignment `x=e` if $WP(\texttt{x=e}, \mathcal{E}(b)) = \mathcal{E}(b)$.

Third, for each computation $\mathcal{F}_V(\varphi)$, we perform an analysis to produce a set $V' \subseteq V$, such that $\mathcal{E}(V')$ contains all predicates from $\mathcal{E}(V)$ that can possibly be part of a cube that implies φ. Therefore, $\mathcal{F}_V(\varphi)$ can safely be replaced by $\mathcal{F}_{V'}(\varphi)$, reducing the number of cubes to explore. This set V' is determined by a syntactic cone-of-influence computation. Starting with an empty set E' we find predicates in $\mathcal{E}(V)$ that mention a location or an alias of a location in φ, add these predicates to E', determine the set of locations mentioned in these predicates, and iterate until reaching a fixpoint. $V' \subseteq V$ is the set of boolean variables such that $\mathcal{E}(V') = E'$.

Fourth, we try several syntactic heuristics to construct $\mathcal{F}_V(\varphi)$ directly from φ. As a simple example, if there exists a boolean variable b such that $\mathcal{E}(b) = \varphi$, then we return b, without requiring any calls to the theorem prover. Fifth, we cache all computations by the theorem prover and the alias analysis, so that work is not repeated.

While the worst-case complexity of computing the abstraction is exponential in the number of predicates, the above optimizations dramatically reduce the number of calls made to the theorem prover in most examples. Moreover, the above optimizations all have the property that they leave the resulting $\mathcal{BP}(P, E)$ semantically equivalent to the boolean program produced without these optimizations. Some of the optimizations described rely on the existence of the `enforce` data invariant for soundness.

If we are willing to sacrifice some precision, there are other optimization opportunities. For example, we can limit the length of cubes considered in the $\mathcal{F}$ computation to some constant k, lowering the $\mathcal{F}$ function's complexity from exponential to $O(n^k)$. In practice, we have found that setting k to 3 provides the needed precision in most cases. As another optimization, we can compute the $\mathcal{F}$ function only on atomic predicates. That is, we recursively convert $\mathcal{F}(\varphi_1 \wedge \varphi_2)$ to $\mathcal{F}(\varphi_1) \wedge \mathcal{F}(\varphi_2)$ and $\mathcal{F}(\varphi_1 \vee \varphi_2)$ to $\mathcal{F}(\varphi_1) \vee \mathcal{F}(\varphi_2)$. This allows us to make use of all of the existing optimizations of the $\mathcal{F}$ function described above in a finer-grained manner. Distribution of $\mathcal{F}$ through $\wedge$ loses no precision, while distribution of $\mathcal{F}$ through $\vee$ can lose precision.

program	lines	predicates	thm. prover calls	runtime (seconds)
floppy	6500	23	5509	98
ioctl	1250	5	500	13
openclos	544	5	132	6
srdriver	350	30	3034	93
log	236	6	98	5

Table 1: The device drivers run through C2BP.

6 Experience

We have implemented C2BP in OCaml, on top of the AST toolkit (a modified version of Microsoft's C/C++ compiler that exports an abstract syntax tree interface to clients), the Simplify [15, 27] and Vampyre [7] theorem provers, and Das's points-to analysis [12].

We have applied C2BP to two problem areas: (1) checking safety properties of Windows NT device drivers, in the context of the SLAM project and the SLAM toolkit; (2) discovering invariants regarding array bounds checking and list-manipulating code.

6.1 The SLAM Toolkit and its Application to NT Device Drivers

The goal of the SLAM project is to automatically check that a program respects a set of *temporal safety* properties of the interfaces it uses. Safety properties are the class of properties that state that "something bad does not happen". An example is requiring that a lock is never released without first being acquired (see [23] for a formal definition). Given a program and a safety property, we wish to either validate that the code respects the property, or find an execution path that shows how the code violates the property.

Given a safety property to check on a C program, the SLAM process has the following phases: (1) abstraction, (2) model checking, and (3) predicate discovery. We have developed the SLAM toolkit to support each of these phases:

- C2BP, which is the topic of this paper;
- BEBOP, a tool for model checking boolean programs [5];
- NEWTON, a tool that discovers additional predicates to refine the boolean program, by analyzing the feasibility of paths in the C program (the subject of a future paper).

The SLAM toolkit provides a fully automatic way of checking temporal safety properties of system software. Violations are reported by the SLAM toolkit as paths over the program P. The toolkit never reports spurious error paths. Instead, it detects such paths and uses them to automatically refine the boolean program abstraction (to eliminate these paths from consideration). Since property checking is undecidable, the SLAM refinement algorithm may not converge. In addition, it may terminate with a "don't know" answer due to the incompleteness of the underlying theorem provers. However, in our experience, it usually converges in a few iterations with a definite answer. One reason for this is that the properties we checked are very control-intensive, and have relatively simple dependencies on data.

We ran the SLAM toolkit on four drivers from the Windows 2000 Driver Development Kit [5], as well as an internally developed floppy device driver, to check for proper usage of locks and proper handling of interrupt request packets (see [6] for the details of the properties checked). The device drivers in the DDK are supposed to be exemplars for others to base their device drivers on. For the two properties we checked, the SLAM toolkit validated these drivers (i.e., found no errors). For the floppy driver under development, the SLAM toolkit found an error in how interrupt request packets are handled.

Table 1 shows the sizes of these drivers, the number of predicates in the predicate input file, the number of theorem prover queries that C2BP made, and the run time for C2BP. For all these examples (and those of the next section), BEBOP ran in under 10 seconds on the boolean program output by C2BP.

6.2 Array Bounds Checking and Heap Invariants

Table 2 shows the results of running C2BP on a set of toy illustrative examples. The program `kmp` is a Knuth-Morris-Pratt string matcher and `qsort` is an array implementation of quicksort, both examples used by Necula [26]. The program `partition` is the list partition example from Figure 1, `listfind` is a list search example, and `reverse` is an example that reverses a list twice. In most cases, the cone-of-influence heuristics in C2BP were able to reduce the number of theorem prover calls to a manageable number. In the case of the `reverse` example, every pair of pointers could potentially alias, and the cone-of-influence heuristics could not avoid the exponential number of calls to the theorem prover.

[5] freely available from `http://www.microsoft.com/ddk/`

program	lines	predicates	thm. prover calls	runtime (seconds)
kmp	75	4	286	7
qsort	45	2	199	5
partition	55	4	263	9
listfind	37	6	4412	172
reverse	73	7	26769	747

Table 2: The array and heap intensive programs analyzed with C2BP.

```
struct node {
  int mark;
  struct node *next;
};
void mark(struct node *list) {
  struct node *this, *tmp, *prev;
  prev = 0;
  this = list;
  /* traverse list and mark, setting back pointers */
  while( this != 0 ) {
    if(this->mark==1)
      break;
    this->mark = 1;
    tmp  = prev;
    prev = this;
    this = this->next;
    prev->next = tmp;
  }
  /* traverse back, resetting the pointers */
  while( prev!=0 ){
    tmp  = this;
    this = prev;
    prev = prev->next;
    this->next = tmp;
  }
}
```

Figure 3: List traversal using back pointers

In our experiments, we were able to construct useful invariants in the code by modeling only a few predicates that occurred in the program. For example, in the array bounds checking examples (`kmp` and `qsort`), where an array a was indexed in a loop by a variable $index$, we simply had to model the bounds $index \geq 0$ and $index \leq length(a)$ in order to produce the appropriate loop invariant. We found that in most cases, the component predicates of the invariant were easy to guess by looking at the conditionals in the programs.

The list reversal example `reverse` is a simplified version of a mark-and-sweep garbage collector. We show the program in Figure 3. In the first *while* loop, the list is traversed in the forward direction, while maintaining back pointers to the previous nodes. In the second loop, the pointers are reversed to get the original list. We wish to verify that the procedure `mark` leaves the shape of the structure unchanged: i.e., for every node h in the list, $h \rightarrow next$ points to the same node before and after the procedure `mark`. To check this, we introduced auxiliary variables `h` and `hnext` into the C code. The variable `h` is chosen non-deterministically to point at any (non-null) element of the list, and `hnext` is initialized with `h->next`. We input the following predicates to C2BP (along with the program of Figure 3):

```
mark {
  h == 0,
  prev == h,
  this == h,
  this->next == hnext,
  prev == this,
  h->next == hnext,
  hnext->next == h
}
```

With this choice of predicates, C2BP constructs an abstract program which is analyzed using BEBOP. BEBOP shows that at the end of the `mark` procedure, $h \rightarrow next = hnext$ holds.

7 Related Work

Our work is inspired by the predicate abstraction work of Graf and Saidi [19]. Predicate abstraction has been used in the verification of cache coherence protocols [13]. However, these efforts work at the specification level, on a language with guarded commands. Doing predicate abstraction on a general-purpose programming language is the novel aspect of our work. A method for constructing abstract models from Java programs has been developed in the Bandera project [17]. Their tool requires the user to provide finite-domain abstractions of data types. Predicate abstraction as implemented in C2BP is more general, as it allows the finite partitioning of a variable's possible values and additionally allows relationships between variables to be defined. Another approach is to use richer type systems to model finite-state abstractions of programs [14].

Shape analysis [30] also uses a form of predicate abstraction, where the predicate language is a first-order logic augmented with transitive closure. In contrast, our predicates are quantifier-free. Shape analysis requires the user to specify how each statement affects each predicate of interest, whereas the C2BP tool computes the abstract transition system automatically using a theorem prover.

Predicate abstraction is a general technique that can be used to add predicate (read "path") sensitivity to program analyses. Ammons and Larus use code duplication followed by a traditional dataflow analysis to achieve path-sensitive results [1]. Bodik and Anik use symbolic back-substitution (i.e., weakest preconditions) followed by value numbering to improve the results of a subsequent three-valued dataflow analysis [8]. The combination of predicate abstraction by C2BP and path-sensitive dataflow analyses in BEBOP could be used to achieve similar results.

Prior work for generating loop invariants has used symbolic execution on the concrete semantics, augmented with widening heuristics [32, 33]. The Houdini tool guesses a candidate set of annotations (invariants) and uses the ESC/Java checker to refute inconsistent annotations until convergence [18]. In contrast, the tools C2BP and BEBOP use a combination of abstraction (from C program to boolean program) and iterative analysis of the abstracted C program to find loop invariants expressible as boolean functions over a given set of predicates.

8 Conclusions

We summarize our main contributions:

- C2BP is the first predicate abstraction tool that works on a general-purpose programming language.
- We have taken efforts to handle features such as procedures and pointers in a sound and precise way.
- We have explored several optimizations to reduce the number of calls made to the theorem prover by C2BP.
- We have demonstrated the use of C2BP on programs from varying domains — device drivers, array-manipulating programs, and pointer-manipulating programs.

Though we fully support pointers in C2BP, our predicates are quantifier-free. Stating certain properties of unbounded data structures may require a more expressive logic. For this purpose, it would be interesting to enrich the predicate language with dependent types and recursive types. Among other things, the aliasing problem becomes more complicated in this setting. For example, if T is a type that denotes lists of even length, then the predicate $(p \in T)$ is **true** if p points to an object of type T. Consider an assignment of the form `q->next = NULL`. To update $(p \in T)$, we have to consider the possibility that q can point anywhere inside the list pointed to by p.[6] One way around this difficulty is to use linear types to encode that there are no external pointers to the list other than p. It would also be interesting to investigate the use of predicates expressible in some recent pointer logics [29, 22].

We have focused on predicate abstraction of single-threaded programs, and it would be interesting to extend C2BP to work for multi-threaded code. Several issues need to be resolved here. First, one needs to establish an appropriate notion of atomicity of execution. Next, while abstracting any statement one has to account for the possibility of interference from another thread. Even if such an abstraction were possible, model checking boolean programs with even two threads is undecidable. One possible solution is to further abstract boolean programs to finite-state machines, and then use traditional model checking algorithms to explore interleaving executions of the finite-state machines. A further problem is that in certain situations, it is not possible to know the number of threads in advance. If we were to first abstract boolean programs to finite-state machines, then it is possible to use parameterized model checking to handle an arbitrary number of threads [2]. It is not clear if these abstractions can be performed automatically.

We have chosen C as our source language for predicate abstraction. However, our fundamental contribution is a set of techniques to handle procedure calls and pointers during predicate abstraction. The techniques in this paper can be adapted to construct predicate abstractions of programs written in other imperative languages such as Java.

We plan to improve some inefficiencies we have in the implementation. The theorem prover is currently started as a separate process each time it is used, which is very inefficient. A more fundamental issue is that we currently use theorem provers as black boxes. We plan to investigate if opening up the internals of the theorem prover can improve the efficiency of the abstraction process.

[6] We thank Frank Pfenning for this observation.

Generating predicates for a predicate abstraction tool like C2BP is another open research problem. We are currently building a tool called NEWTON in the SLAM toolkit to generate predicates from the model checker's counterexamples, using path simulation. We are also exploring predicate generation using value flow analysis on the program, with respect to the properties of interest. Our current approach seems to work as long as the properties of interest have relatively simple dependencies on data. For data-intensive properties, predicate generation may have to use widening heuristics as in [32, 33].

Acknowledgements. We thank Andreas Podelski for helping us describe the C2BP tool in terms of abstract interpretation. We thank Manuvir Das for providing us his one-level flow analysis tool. We thank the developers of the AST toolkit at Microsoft Research, and Manuel Fähndrich for providing us his OCaml interface to the AST toolkit. We thank Craig Chambers for several interesting discussions about C2BP. Thanks also to the members of the Software Productivity Tools research group at Microsoft Research for many enlightening discussions on program analysis, programming languages and device drivers, as well as their numerous contributions to the SLAM toolkit.

References

[1] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *PLDI 98: Programming Language Design and Implementation*, pages 72–84. ACM, 1998.

[2] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031. Springer-Verlag, 2001.

[3] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR Technical Report 2001-10, Microsoft Research, 2000.

[4] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031. Springer-Verlag, 2001.

[5] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[6] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001: SPIN Workshop*, LNCS 2057, May 2001.

[7] D. Blei and et al. Vampyre: A proof generating theorem prover — http://www.eecs.berkeley.edu/˜ rupak/vampyre.

[8] R. Bodik and S. Anik. Path-sensitive value-flow analysis. In *POPL 98: Principles of Programming Languages*, pages 237–251. ACM, 1998.

[9] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[10] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *ICSE 00: Software Engineering*, 2000.

[11] P. Cousot and R. Cousot. 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*, pages 238–252. ACM, 1977.

[12] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.

[13] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV 00: Computer-Aided Verification*, LNCS 1633, pages 160–171. Springer-Verlag, 1999.

[14] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation*. ACM, 2001.

[15] D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover – http://research.compaq.com/src/esc/simplify.html.

[16] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[17] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: Software Engineering (to appear)*, 2001.

[18] C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Information Processing Letters (to appear)*, 2001.

[19] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[20] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[21] N. Heintze. Set-based analysis of ML programs. In *LFP 94: LISP and Functional Programming*, pages 306–317. ACM, 1994.

[22] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL 01: Principles of Programming Languages*, pages 14–26. ACM, 2001.

[23] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[24] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural side effect analysis with pointer aliasing. In *PLDI 93: Programming Language Design and Implementation*, pages 56–67. ACM, 1993.

[25] J. M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, Lecture Notes of an International Summer School, pages 25–34. D. Reidel Publishing Company, 1982.

[26] G. Necula. Proof carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.

[27] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[28] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[29] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*, pages 303–321. Palgrave, 2001.

[30] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.

[31] M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[32] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *POPL 77: Principles of Programming Languages*, pages 132–143. ACM, 1977.

[33] Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *PLDI 00: Programming Language Design and Implementation*, pages 70–82. ACM, 2000.