Predicate Abstraction for Software Verification

Cormac Flanagan Shaz Qadeer
Compaq Systems Research Center
130 Lytton Ave, Palo Alto, CA 94301

Abstract

Software verification is an important and difficult problem. Many static checking techniques for software require annotations from the programmer in the form of method specifications and loop invariants. This annotation overhead, particularly of loop invariants, is a significant hurdle in the acceptance of static checking. We reduce the annotation burden by inferring loop invariants automatically.

Our method is based on predicate abstraction, an abstract interpretation technique in which the abstract domain is constructed from a given set of predicates over program variables. A novel feature of our approach is that it infers universally-quantified loop invariants, which are crucial for verifying programs that manipulate unbounded data such as arrays. We present heuristics for generating appropriate predicates for each loop automatically; the programmer can specify additional predicates as well. We also present an efficient algorithm for computing the abstraction of a set of states in terms of a collection of predicates.

Experiments on a 44KLOC program show that our approach can automatically infer the necessary predicates and invariants for all but 31 of the 396 routines that contain loops.

1 Introduction

Ensuring the reliability of software systems is important but extremely difficult, due to the test coverage problem. Static analysis techniques could potentially verify that a program satisfies a (lightweight) specification, for example, that it should not crash. A completely automatic solution to this checking problem is impossible in general since the problem is undecidable. Semi-automatic techniques rely on the programmer to provide additional annotations describing method specifications and loop invariants. This annotation overhead is a significant obstacle to the use of formal techniques in software design. While method specifications also function as useful documentation and may be helpful for code maintenance, loop invariants do not provide comparable benefits along these lines. Thus, the task of writing appropriate loop invariants for a large program is perceived to be especially tedious.

In this paper, we present a new method for automatically inferring loop invariants. Our method is based on *predicate abstraction* [GS97], which is a special form of abstract interpretation [CC77] in which the abstract domain is constructed using a given set of predicates. These predicates are generated in a heuristic manner from the program text; the programmer is free to provide additional predicates. Given a suitable set of predicates for a given loop, our algorithm infers loop invariants that are boolean combinations of these predicates. Thus, the problem of computing loop invariants is reduced to the easier problem of guessing a relevant set of simple predicates.

Our algorithm analyzes the loops of a method in order, since knowing the invariant for one loop constrains the possible initial states of a subsequent loop, and helps infer a more precise invariant for that loop. The invariant for each loop is computed by iterative approximation. The first approximation is obtained by abstracting the set of reachable states at loop entry. Each successive approximation enlarges the current approximation to include the states reachable by executing the loop body once from the states in the current approximation. The iteration terminates in a loop invariant since the abstract domain is finite.

The fundamental operation in the algorithm described above is the abstraction of a set of reachable states. The abstraction operation is performed by making queries to an automatic theorem prover, and may require an exponential number of queries. Existing methods [DDP99, SS99] perform much better in practice, but are still costly. We describe a new abstraction algorithm that requires fewer queries than existing algorithms, particularly for large numbers of predicates.

The loop invariant inference method described above is inadequate for verifying loops that manipulate unbounded data such as arrays. For such loops, the required invariants are universally quantified. Naive application of predicate abstraction requires the user to guess the full quantified expression as a predicate, which is no simpler than guessing the loop invariant itself. We avoid this limitation by allowing the predicates to refer to *skolem constants*, which are fresh variables not mentioned anywhere else in the program. These skolem constants represent some fixed, unknown value, thus allowing us to universally quantify them out from the inferred invariant without loss of soundness.

We have implemented our algorithm using the infrastructure of the Extended Static Checker for Java (ESC/Java) [DLNS98, LSS99]. The goal of ESC/Java is to detect statically programming errors that are normally detected only at run-time, if ever. Such errors include null pointer dereferences, array bounds errors, division by zero, and the violation of programmer-specified properties such as method specifications and object invariants. We have evaluated our method on several small but interesting examples, including the list partition algorithm from the SLAM paper [BMMR01] and a selection sort example. We have also evaluated our method on the front end for ESC/Java, which contains about 44K lines of Java code and 520 loops. Our loop invariant inference algorithm, using heuristics we develop for guessing candidate predicates, was able to verify the correctness of all but 31 of the 396 methods that contain loops.

Background. Beginning in the 1970's, a number of researchers have addressed the problem of automatically inferring loop invariants. The first papers [GW74, Weg74, KM76] proposed heuristics for generating loop invariants for sequential programs. Algorithms based on iterative forward and backward traversal, with heuristics for terminating the iteration, appeared soon after [GW75, SI77]. Abstract interpretation [CC77] introduced the *widening* operator, which was used to compute [CH78] fixpoints more systematically. ESC/Modula-3 [DLNS98] experimented with generating loop invariants using both abstract interpretation and a second technique called loop modification inference. In the 1990's, a number of researchers worked on generating auxiliary invariants for proving safety properties of concurrent algorithms and protocols [MP92, BLS96, BBM97].

Predicate abstraction has also been used for verifying protocols [DDP99, SS99] and hardware [CGJ$^+$00]. Recently the SLAM project [BMMR01] has also explored predicate abstraction for sequential programs. They perform abstraction on each program statement, whereas our method abstracts only when necessary, that is, at loop headers. As a result, our inferred invariants are more precise and we require fewer predicates. In addition, the SLAM approach is unable to infer universally-quantified invariants, which we have found crucial for verifying many programs.

```
/*@ requires a != null && b != null */
/*@ requires a.length == b.length */
/*@ ensures  \result == a.length || b[\result] */

int find(int[] a, boolean[] b) {
    int spot = a.length;
    for (int i = 0; i < a.length; i++) {
        if (spot == a.length && a[i] != 0)
            spot = i;
        b[i] = (a[i] != 0);
    }
    return spot;
}
```

Figure 1: Method `find`

Outline. Section 2 motivates our approach with a simple example. Section 3 describes an idealized programming language and Section 4 describes how to infer loop invariants for this language. Our inference algorithm depends on the predicate abstraction algorithm described in Section 5. Section 6 describes our heuristics for guessing suitable predicates. Section 7 reports on our experience using our method on various examples. Finally, we conclude in Section 8.

2 Motivating example

We illustrate our technique for inferring loop invariants using the method `find` shown in Figure 1. This method takes as parameters an array of integers `a` and an array of booleans `b`. The method returns the index of the first nonzero element of `a` if one exists and `a.length` otherwise. The method also sets the i-th element of `b` to true if the i-th element of `a` is nonzero, and to false otherwise. The preconditions of the method specified by the annotation `/*@ requires .. */` states that the arrays are nonnull and of the same length. The postcondition `/*@ ensures .. */` states that the returned index (denoted by `\result`) is either `a.length` or `b` is true at that index.

The method body consists of a loop that iterates over the elements of `a` and sets the elements of `b` appropriately. A variable `spot`, initialized to `a.length`, is set to the index of the first nonzero element if one exists. Proving that no array bounds violation occurs requires the loop invariant `0 <= i`. This is a simple invariant involving a single predicate and can be stated without much effort. Proving the postcondition of this method requires the more complex loop invariant:

```
/*@ loop_invariant spot == a.length ||
                  (b[spot] && spot < i) */
```

Constructing this loop invariant requires ingenuity in choosing the correct boolean combination of the three predicates `spot == a.length`, `b[spot]`, and `spot < i`. Predicate abstraction allows us to specify only the predicates (none of which is a loop invariant by itself), and derive the loop invariant automatically from them. These loop predicates are suggested to the algorithm by the annotation:

```
/*@ loop_predicate spot == a.length, b[spot], spot < i */
```

The predicate abstraction algorithm computes the strongest boolean combination of these three predicates by iterative approximation. The first approximation is `spot == a.length`, which holds at loop entry. The second approximation, after one loop iteration, is `spot == a.length || (b[spot] && spot < i)`. This approximation is a fixpoint and is therefore the desired loop invariant.

Now suppose we want to verify the following additional postcondition, which states that every element of the array `b` before the returned value contains `false`:

```
/*@ ensures (\forall int j; 0 <= j && j < \result
                         ==> b[j] == false) */
```

Proving this postcondition requires the additional loop invariant:

```
/*@ loop_invariant
    (\forall int j; 0 <= j && j < i && j < spot
              ==> b[j] == false) */
```

The loop invariant is universally quantified; the component predicates of the invariant refer to the quantified variable `j` as well to the program variables. Therefore, in order to use predicate abstraction as before, we have to specify the whole loop invariant as a predicate, which does not simplify the problem. We get around this difficulty by introducing a skolem constant `j`, which is a fresh variable not mentioned elsewhere in the program, and by introducing extra predicates that refer to the skolem constant `j` as well as to the program variables.

```
/*@ skolem_constant int j;
    loop_predicate 0 <= j, j < i, j < spot, b[j]; */
```

We now perform predicate abstraction yielding the new invariant:

```
0 <= j && j < i && j < spot ==> b[j] == false
```

Since the skolem constant `j` represents some fixed unknown value, this invariant is valid for any value of `j`. Therefore, the skolem constant can be safely quantified out to yield the desired universally-quantified loop invariant. We have found this technique to be crucial when checking programs that manipulate unbounded data, such as arrays.

$$
\begin{array}{rcll}
A, B & \in & \mathit{Stmt} & ::= \ \textbf{assert}\ e \\
 & & & \ \ \mid\ \ \textbf{assume}\ e \\
 & & & \ \ \mid\ \ x := e \\
 & & & \ \ \mid\ \ A \,;\, B \\
 & & & \ \ \mid\ \ A \,\Box\, B \\
 & & & \ \ \mid\ \ \{\mathcal{P}, I\}\ \textbf{while}\ e\ \textbf{do}\ B\ \textbf{end} \\
x & \in & \mathit{Var} & \text{(variables)} \\
e & \in & \mathit{Expr} & \text{(expressions)} \\
I & \in & \mathit{Formula} & \text{(logical formulae)} \\
\mathcal{P} & \subseteq & \mathit{Formula} & \text{(loop predicates)}
\end{array}
$$

Figure 2: A guarded command language

3 A guarded command language

We have implemented our technique as part of ESC/Java. To verify a Java program, ESC/Java translates each Java method and its specification into a logical formula called a verification condition (VC). Ideally, the VC has the property that if it is valid then the method is correct, *i.e.*, it implements its specification and never performs an erroneous operation such as dereferencing a null pointer.

Deriving verification conditions for a large and realistic language such as Java is quite complex. To help structure and modularize this translation, ESC/Java first translates each method and its specification into an intermediate representation, known as a guarded command. This translation eliminates many of the complexities of the Java programming language; it is outlined elsewhere [LSS99]. In this paper, we concentrate on the subsequent task of inferring suitable invariants for the loops in the intermediate representation.

We base our development on the intermediate language shown in Figure 2. The language is a variation of Dijkstra's guarded commands [Dij76], together with some more recent additions (see, e.g., [Nel89, BvW98]). The language includes assert and assume statements, assignment statements, sequential composition, nondeterministic choice, and loops. By including assume statements, we no longer need the "guards" that originally gave the language its name.

Expressions in the language are intentionally unspecified, since their structure is mostly irrelevant to our development. We assume that the set *Expr* contains expressions that are pure (side-effect free), and include at least the boolean constants *true* and *false*. The set *Formula* of logical formulae is an extension of boolean expressions that includes at least the usual boolean operators ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $=$) and quantification, and is closed under substitution of expressions for variables. We let *Var* denote the set of program variables, and use

S	$Norm(Q, S)$	$Wrong(Q, S)$
$x := e$	$\exists x'.\ x = e(x \leftarrow x') \wedge Q(x \leftarrow x')$	$false$
assert P	$Q \wedge P$	$Q \wedge \neg P$
assume P	$Q \wedge P$	$false$
$A \ \square\ B$	$Norm(Q, A) \vee Norm(Q, B)$	$Wrong(Q, A) \vee Wrong(Q, B)$
A ; B	$Norm(Norm(Q, A), B)$	$Wrong(Q, A) \vee Wrong(Norm(Q, A), B)$
$\{\mathcal{P}, I\}$ **while** e **do** B **end**	$Norm(Q, desugar(S))$	$Wrong(Q, desugar(S))$

Figure 3: Strongest-postcondition semantics

$Q(x \leftarrow e)$ to denote the capture-free substitution of e for every free occurrence of x in a formula Q. Where appropriate, we use double-quotes "..." to distinguish constructed syntax from mathematical definitions and algorithms.

3.1 Informal semantics

The execution of a statement may either terminate normally or it may "go wrong" due to a failed assertion. The execution of the statement "**assert** e" terminates normally if the condition e evaluates to *true* in the current program state, and goes wrong otherwise. The assume statement is partial: "**assume** e" terminates normally if the condition e evaluates to *true*, and simply cannot be executed from a state where e evaluates to *false*. The assignment statement "$x := e$" updates the program state so that x is bound to the current value of the expression e. The statement "A ; B" denotes the sequential composition of A and B. The execution of the choice statement "$A \ \square\ B$" executes either A or B, but the choice between the two is made arbitrarily. The while loop "$\{\mathcal{P}, I\}$ **while** e **do** B **end**" has the conventional semantics that B is executed as long as the condition e remains *true*. The loop invariant I is provided to aid in the verification process; it is required to hold at the beginning of each loop iteration. The set $\mathcal{P}$ of predicates is used for the automatic inference of loop invariants.

All variables have arbitrary values in a program's initial state. The indeterminism arising from choice statements and from the program's initial (arbitrary) state can be tamed by assume statements: the semantics of a program considers only those executions in which the condition of each executed assume statement evaluates to *true*. For example, the statement "(**assume** e ; A) $\square$ (**assume** $\neg e$; B)" is the deterministic statement commonly written as "**if** e **then** A **else** B **end**".

3.2 Formal semantics

We formally define the semantics of our language using the strongest postcondition translations

$$Norm, Wrong : Formula \times Stmt \rightarrow Formula$$

shown in Figure 3. For an execution of S that starts in an initial state satisfying the formula Q, the postcondition $Norm(Q, S)$ characterizes post-states in which that execution could terminate normally. The postcondition $Wrong(Q, S)$ characterizes the post-states in which that execution could go wrong by failing an assert.

The definition of *Norm* and *Wrong* is straightforward. The normal postcondition $Norm(Q,\ x := e)$ of an assignment statement with respect to a precondition Q is $\exists x'.x = e(x \leftarrow x') \wedge Q(x \leftarrow x')$; this formula holds in the post-state of the assignment provided there exists some x' representing the initial value of x such that Q holds in that initial state and x contains the results of evaluating e in that initial state. An assignment statement can never go wrong, and thus its wrong postcondition is *false*.

The strongest postcondition translation for a while loop relies on an auxiliary function that desugars loops into more primitive statements. This desugaring relies on the loop invariant I which is required to hold at the beginning of each iteration of the loop; it is defined as:

$$\begin{array}{l} desugar(\text{"}\{\mathcal{P}, I\}\ \textbf{while}\ e\ \textbf{do}\ B\ \textbf{end}\text{"}) = \\ \quad \textbf{assert}\ I\ ;\ havoc(targets(B))\ \ ;\ \textbf{assume}\ I\ ; \\ \quad (\quad (\textbf{assume}\ e\ ;\ B\ ;\ \textbf{assert}\ I\ ;\ \textbf{assume}\ false) \\ \quad\ \ \square\ \ \textbf{assume}\ \neg e) \end{array}$$

The function $targets : Stmt \rightarrow 2^{Var}$ returns the set of variables assigned in the loop body. The function $havoc : 2^{Var} \rightarrow Stmt$ assigns arbitrary values to these variables; it is defined by

$$havoc(\{x_1, \ldots, x_n\}) = \text{"}x_1 := y_1\ \ ;\ \ldots\ ;\ x_n := y_n\text{"}$$

where $y_1, \ldots, y_n$ are fresh variables that hold arbitrary values (from the program's initial state).

The desugared code ensures that the loop invariant I holds initially, and then sets the loop targets (the variables modified by the loop body) to arbitrary values that satisfy the loop invariant. The code then checks that if e is true, then the loop invariant still holds after executing B; if e is false, then the desugared loop terminates (and execution continues at the subsequent statement).

An important aspect of this desugaring is that the invariant need only explicate properties of loop targets, since properties of other variables that held in the pre-state of the loop will be known to still hold in the loop body and after loop termination.

To show that a statement S cannot go wrong from any initial state, it suffices to prove that the verification condition[1]

$$\neg(\mathit{Wrong}(\mathit{true}, S))$$

is valid, for example, by using an automatic theorem prover [Nel81].

4 Inferring loop invariants

Verifying a program using the techniques outlined in the previous section requires that each loop of the program is first annotated with a suitable loop invariant. Our experience with ESC/Java indicates that the burden of specifying loop invariants is substantial. In this section, we describe an algorithm for inferring such invariants automatically.

Our inference algorithm processes the loops of a program in order, since knowing the invariant for one loop constrains the possible initial states of a subsequent loop, and helps infer a precise invariant for that loop. This in-order processing is facilitated by our use of strongest postconditions (as opposed to the more common weakest preconditions). The procedure *traverse*, defined in Figure 4, takes as input a statement S, and returns a modified version of S that includes an inferred invariant for each loop in S. The procedure also takes as input a context C, which is the code preceding S, and hence constrains the initial states for the execution of S. We assume suitable invariants have been inferred for any loops in C.

If S is an assert, assume, or assignment statement, then it does not contain any loops, and *traverse* returns S unmodified. If S is a choice statement, then its sub-statements are processed recursively. Similarly, if S is a sequential composition A ; B, the sub-statements A and B are also processed recursively, with A' (the version of A with inferred invariants) being appended to the context used when inferring invariants for B. For the interesting case where S is a while loop, the helper procedure *infer* (see Figure 4) is called to infer a suitable invariant for that loop.

```
Stmt traverse(Stmt C, Stmt S) {
  case S of {
     "x := e"       → { return S; }
     "assert P"     → { return S; }
     "assume P"     → { return S; }
     "A □ B"        → { A' = traverse(C, A);
                        B' = traverse(C, B);
                        return "A' □ B' "; }
     "A ; B"        → { A' = traverse(C, A);
                        B' = traverse("C ; A'", B);
                        return "A' ; B' "; }
     "{𝒫, I} while e do B end" → {
        ⟨J, B'⟩ = infer(C, S);
        return "{𝒫, I ∧ J} while e do B' end"
     }
  }
}

⟨Formula, Stmt⟩ infer(Stmt C, Stmt S) {
  let "{𝒫, I} while e do B end" = S;
  Stmt H = havoc(targets(B));
  AbsDomain r = α(Norm(true, C));
  while (true) {
     Formula J = γ(r);
     Stmt A = "assume e ∧ I ∧ J ";
     Stmt B' = traverse("C ; H ; A", B);
     Formula Q = Norm(true, "C ; H ; A ; B' ");
     AbsDomain next = r ∨ α(Q);
     if (next = r) return ⟨J, B'⟩;
     r = next;
  }
}
```

Figure 4: Procedures *traverse* and *infer*

The required invariant for S could, in theory, be computed iteratively, by repeatedly applying the strongest postcondition transformer to the loop body. However, because the set of possible program states is infinite, in most cases this iterative algorithm does not converge.

To solve this convergence problem, we use the predicates $\mathcal{P} = \{p_1, \ldots, p_n\}$ associated with the loop to abstract the infinite concrete state space to a finite *abstract domain*. This abstract domain is the set of all boolean functions of n boolean variables $\mathcal{B} = \{b_1, \ldots, b_n\}$, where each boolean variable b_i corresponds to the predicate p_i. An abstract domain element f is thus a boolean function over $\mathcal{B}$, and represents the con-

[1] ESC/Java actually uses a more efficient algorithm for generating verification conditions that avoids the exponential blowup associated with the standard weakest precondition and strongest postcondition translations [FS01].

crete states described by the *concretization*

$$\gamma(f) = f(b_1 \leftarrow p_1, \ldots, b_n \leftarrow p_n)$$

which replaces each variable b_i in f by the corresponding predicate p_i.

Conversely, for any predicate Q over the concrete state space, the corresponding abstract domain element is obtained via the *abstraction* $\alpha(Q)$, which is the strongest boolean function on $\mathcal{B}$ such that

$$Q \Rightarrow \gamma(\alpha(Q)).$$

Section 5 describes an efficient algorithm for computing $\alpha(Q)$; the remainder of this section describes how to use the abstraction and concretization operations to converge on a suitable loop invariant.

The procedure *infer* takes as arguments a loop context C and a loop S. The procedure keeps track of an abstraction r of the set of reachable states. The set of reachable states is initially $Norm(true, C)$, and so r is initialized with the corresponding abstraction. The procedure iteratively analyzes the loop body and updates r until a fixpoint is reached.

At each iteration, a concrete representation of the current set of reachable states is generated in J. The statement A explicates assumptions that hold at the beginning of the loop body. These assumptions include the loop guard e, the supplied loop invariant I, and the current approximation J to the invariant being inferred. The resulting context for the loop body B is then "C ; H ; A", where H havocs the targets of B. Suitable invariants for nested loops in B are computed by recursively calling *traverse* on B with the new context, yielding a modified loop body B'. Applying the normal postcondition operator to "C ; H ; A ; B' " yields a predicate Q approximating the set of states that are reachable at the end of the analyzed loop. We extend r with the corresponding abstraction $\alpha(Q)$ and iterate. Eventually a fixpoint is reached, and the invariant $J = \gamma(r)$ is then a valid invariant for the loop. This invariant, together with the modified loop body B', is returned as the result of *infer*.

The following theorem states that the loop invariant computed by *infer* is correct; that is, it holds on entry to the loop and also holds after an arbitrary loop iteration.

Theorem 4.1 (Correctness of *infer*) *Suppose the procedure call infer*$(C$, "$\{\mathcal{P}, I\}$ **while** e **do** B **end**"$)$ *returns the tuple* $\langle J, B'\rangle$. *Let* $H = havoc(targets(B))$. *Then the following are valid:*

$$Norm(true, C) \Rightarrow J$$
$$Norm(true, \text{“}C \; ; H \; ; \textbf{assume } e \wedge I \wedge J \; ; B'\text{”}) \Rightarrow J$$

Our correctness condition for *traverse* is that if the supplied loop invariants in a statement S are correct (although possibly not sufficient to verify the assertions in S), then the loop invariants inferred by *traverse* are also correct. To formalize the notion of an incorrect loop invariant, we introduce a function $h : Stmt \rightarrow Stmt$ that removes all assertions from the given statement. Thus $h(S)$ behaves exactly like S, except that it never goes wrong due to a failed assertion; if $h(S)$ goes wrong, it must be due to an incorrect loop invariant. Thus, the loop invariants of S are *correct* if the verification condition $\neg(Wrong(true, h(S)))$ is valid.

Theorem 4.2 (Correctness of *traverse*) *Suppose S is a statement and $S' = traverse(S$, "**assume** true"$)$. If the loop invariants of S are correct then the loop invariants of S' are correct.*

4.1 Inferring quantified loop invariants

The algorithm outlined above infers loop invariants that are boolean combinations of the given predicates. In many situations, such loop invariants are insufficient, particularly for loops that manipulate unbounded data such as arrays. For example, verifying that a loop clears an array requires a loop invariant stating that all elements in the array up to the current index are zero. Automatically verifying such loops requires the ability to infer *universally-quantified* loop invariants.

This section describes how to extend our approach to infer such universally-quantified loop invariants. We achieve this by allowing the predicates to refer to *skolem constants*, which are fresh variables not mentioned elsewhere in the program. We use $\mathcal{S}$ to denote the set of skolem constants, and use *SCFormula* to denote formulae that may contain references to these constants. The candidate predicates $\mathcal{P}$ accompanying each loop may now include skolem constants, i.e., $\mathcal{P} \subseteq SCFormula$.

To infer universally-quantified loop invariants, we first proceed as outlined earlier. The concretization operation of the procedure *infer*:

$$J = \gamma(r);$$

returns a boolean combination of the predicates, which now includes references to skolem constants. Since these skolem constants do not appear elsewhere in the program, they are simply variables with some fixed, unknown value, and we can universal quantify over them without loss of soundness Thus, we replace the above assignment with:

$$J = \text{“}\forall \mathcal{S}.\gamma(r)\text{”};$$

This universally-quantified loop invariant is used when analyzing subsequent iterations of the loop, and it is also returned as the inferred invariant once the fixpoint is reached. No other changes are required, and the correctness arguments of Theorems 4.1 and 4.2 still hold. This ability to infer universally-quantified loop invariants is often crucial, as illustrated by the examples in Section 7.

5 Predicate abstraction

The invariant inference algorithm relies on an implementation of the abstraction operation $\alpha(Q)$, which is the focus of the present section.

We begin by introducing some useful terminology. Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the given set of predicates, and let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be the corresponding set of boolean variables. A *literal* l is either b_i or $\neg b_i$ for some $1 \le i \le n$. A *clause* d is a set of literals in which each boolean variable appears at most once. A clause of size n thus mentions all boolean variables, and is called a *maximal clause*. The meaning of a clause is the *disjunction* of its literals. The *size* of a clause is its cardinality. We extend the usual boolean operations to clauses.

Recalling the definition of the previous section, the abstraction operation $\alpha(Q)$ returns the strongest boolean function f over the boolean variables such that $Q \Rightarrow \gamma(f)$. The abstraction $\alpha(Q)$ is computed using an automatic theorem prover; a sequence of validity queries are used to identify the relationship between Q and various boolean combinations of the predicates p_i. A naive implementation of the abstraction operation, requiring 2^n validity queries, is:

$$\alpha(Q) = \bigwedge \{d \mid d \text{ is a maximal clause and } \ Q \Rightarrow \gamma(d)\}$$

The procedure *Union* adapts this algorithm to the computation of the abstract domain element $r \vee \alpha(Q)$, which is required by *infer*. As an optimization, when computing $r \vee \alpha(Q)$, we only need to consider clauses that are implied by r.

```
AbsDomain Union(AbsDomain r, Formula Q) {
    AbsDomain result = true;
    for each maximal clause m {
        if ( (r ⇒ m) ∧ (Q ⇒ γ(m)) )
            result = result ∧ m;
    }
    return result;
}
```

The abstract domain element computed by this algorithm is a conjunction of clauses of length n. Our experience with predicate abstraction indicates that the required abstract domain element can often be expressed as a conjunct of much smaller clauses, where the length of each clause is typically at most 3. We use this insight to optimize the above algorithm. In particular, whenever we find a maximal clause m such that $r \Rightarrow m$ and $Q \Rightarrow \gamma(m)$, we try to shrink m to a stronger (smaller) clause c such that c also enjoys the property $r \Rightarrow c$ and $Q \Rightarrow \gamma(c)$. This strengthening operation can be performed in a greedy manner, by starting with c equal to m, and iteratively dropping as many literals as possible from c while preserving this property. Having derived a stronger clause c, we can conjoin it to *result*, and subsequently skip consideration of all maximal clauses m that are implied by *result*. The following algorithm refines the previous one with these ideas.

```
AbsDomain Union(AbsDomain r, Formula Q) {
    AbsDomain result = true;
    for each maximal clause m {
        if ( (result ⇏ m) ∧ (r ⇒ m) ∧ (Q ⇒ γ(m)) ) {
            c = m;
            for each literal l in m {
                d = c \ {l};
                if ( (r ⇒ d) ∧ (Q ⇒ γ(d)) ) { c = d; }
            }
            result = result ∧ c;
        }
    }
    return result;
}
```

As a final improvement, we replace the iterative algorithm that shrinks m to a stronger clause c such that $r \Rightarrow c$ and $Q \Rightarrow \gamma(c)$ with a divide-and-conquer algorithm. This algorithm splits m into two clauses m_1 and m_2 such that $m = m_1 \vee m_2$. If m_2 satisfies the properties $r \Rightarrow m_2$ and $Q \Rightarrow \gamma(m_2)$ then we ignore m_1 and proceed to extract c from m_2. However, if m_2 does not satisfy these properties, then we first recursively extract from m_1 a stronger clause c_1 such that $r \Rightarrow c_1 \vee m_2$ and $Q \Rightarrow \gamma(c_1 \vee m_2)$; we then recursively extract from m_2 a stronger clause c_2 such that $r \Rightarrow c_1 \vee c_2$ and $Q \Rightarrow \gamma(c_1 \vee c_2)$, thus yielding the resulting clause $c = c_1 \vee c_2$ with the desired property. Although this algorithm may require $O(n.2^n)$ theorem prover queries in the worst-case, in practice it performs quite well (see Section 7.4).

Our implementation of this algorithm uses a dual representation for the abstract domain. Each abstract domain element is represented as a binary decision diagram (BDD) [Bry86], thus allowing the implication tests on abstract domain elements ("$result \not\Rightarrow m$", "$r \Rightarrow m$", and "$r \Rightarrow d$") to be performed efficiently. In addition, we also represent abstract domain elements as conjuncts-of-clauses. The procedure *union* naturally computes its results as a conjunct-of-clauses. By preserving this structure, we can present the inferred invariant for each loop as a conjuncts of (typically small) clauses, each of which can be comprehended as a separate invariant by the programmer. This presentation is more comprehensible than the BDD representation for all the inferred invariants we have inspected.

5.1 Related work

We know of two other predicate abstraction algorithms. The first method [DDP99] uses a binary decision tree where each vertex at depth k represents a clause of size k. The clause at a vertex is a superset of the clause at its parent. The ordering of the variables in the decision tree is fixed using some heuristic. While computing $\alpha(Q)$, if there is a clause d at a vertex such that $Q \Rightarrow \gamma(d)$, then the clauses at the descendants of that vertex can be ignored. Thus, this algorithm also tries to find small clauses, but its ability to do so is strongly constrained by the fixed variable ordering. This procedure may require $O(2^{n+1})$ theorem prover queries.

The second method [SS99] generates all clauses in increasing order of size. Again, if $Q \Rightarrow \gamma(d)$ for some clause d, then the algorithm ignores all clauses d' such that $d \subset d'$. This algorithm also searches for small clauses that are implied by Q but this search may require enumerating excessively many clauses (up to 3^n clauses). Thus, this algorithm may require $O(3^n)$ theorem prover queries.

Our algorithm does not suffer from the ordering problem of the binary decision tree algorithm to the same extent. At the same time, it reliably finds small clauses that are implied by Q without enumerating too many clauses. We show experimental results on several examples in Section 7.4 demonstrating that our algorithm performs fewer queries than the two algorithms discussed above.

6 Heuristics for generating predicates

The previous sections reduce the problem of specifying correct invariants to the simpler problem of specifying potentially useful predicates. However, providing such predicates can be still tedious, particularly for large programs. This section presents heuristics for guessing many of these predicates automatically.

When desugaring each loop, ESC/Java first computes a set of *loop targets*, which are the data locations possibly modified by the loop body. A target is expressed in terms of variables that are in scope at the beginning of the loop. Each target is either precise or imprecise. A *precise target* is an l-value: either a variable `x`, a field reference `e.f`, or an array reference $\mathtt{e_1[e_2]}$, and specifies a unique location. An *imprecise target* denotes a set of locations and is of the form `*.f`, $\mathtt{e_1[*]}$, $\mathtt{*[e_2]}$, or `*[*]`, where either an object reference or an array index is unknown. These loop targets are havoced on entry to the loop; we would like to infer invariants that capture relevant properties of the havoced targets.

If the loop targets include the imprecise target $\mathtt{e_1[*]}$, then we would like to infer universally-quantified loop invariants that explicate properties of the modified array $\mathtt{e_1}$. For this purpose, we introduce a skolem constant `sc` of type `int`, together with the predicates `0 <= sc` and `sc < x`, where `x` ranges over integer variables in scope. We also guess predicates about the *skolemized target* expression $\mathtt{e_1[sc]}$, as described below.

For each (precise or skolemized) target expression `t` (of the form `x`, `e.f`, $\mathtt{e_1[e_2]}$, or $\mathtt{e_1[sc]}$), we proceed based on the type of `t`. If `t` is of reference type, we guess that `t` is not `null`. If `t` is an integer, we then guess the predicates `t <= \old(t)`, `t >= \old(t)`, and `t < sc`, where the ESC/Java syntax `\old(t)` refers to the value of `t` on entry to the loop.

To illustrate these heuristics, we briefly describe their application to a loop that clears the array `a`:

```
(for int i=0; i<a.length; i++) a[i] = null;
```

For this loop, the heuristics will generate the predicate `i >= \old(i)`, which is also a loop invariant. Since `\old(i) = 0`, this invariant, together with the loop guard `i < a.length`, is sufficient to verify the absence of array bounds errors. In addition, the heuristics generate three crucial predicates: `0 <= sc`, `sc < i`, and `a[sc] != null`. Predicate abstraction subsequently combines these predicates into the invariant that all array entries below `i` have been cleared:

```
(\forall int sc; 0 <= sc && sc < i ==> a[sc] == null)
```

This invariant is crucial in allowing ESC/Java to verify that after the loop terminates the array contains only `null`s.

These heuristics are targeted towards generating predicates that are useful for the typical proof obligations in ESC/Java. If ESC/Java is used to prove more compilcated properties, these heuristics will have to be augmented. Currently, we do not infer loop invariants stating properties of all elements in a list or a tree, because ESC/Java does not allow convenient expression of the set of objects reachable from an object reference.

7 Implementation and evaluation

We have implemented our method, described in earlier sections, as part of ESC/Java. In this section, we describe the application of our technique to two small illustrative examples, as well as to a substantial program containing over 44K lines of code.

```
Cell partition(Cell l, int v) {
    Cell curr = l, prev = null, newl = null;
    Cell nextCurr;
    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:  //@ assert curr != prev;
            newl = curr;
        } else {
            prev = curr;
        }
        curr = nextCurr;
    }
    return newl;
}
```

Figure 5: Method `partition`

```
/*@ requires a != null;
    ensures (\forall int x, y;
             0 <= x && x < y && y < a.length
         ==> a[x] <= a[y]);
*/

void sort(int[] a) {
    int i = 0;
    while (i < a.length) {
        int k = i, w = a[i], j = i + 1;
        while (j < a.length) {
            if (a[j] < w) {
                k = j;
                w = a[j];
            }
            j = j + 1;
        }
        a[k] = a[i]; a[i] = w;
        i = i + 1;
    }
}
```

Figure 6: Method `sort`

7.1 List partition

Figure 5 shows a list partitioning example adapted from a paper describing the SLAM project [BMMR01]. Each list element is an instance of the class `Cell`, and contains two fields—an integer `val` and a reference `next` to the following cell in the list. The method `partition` takes two arguments, a list `l` and an integer `v`. It removes every cell with value greater than `v` from `l` and returns a new list containing all those cells.

The SLAM tool can verify, using their predicate abstraction algorithm, that at the control point `L` the variable `curr` is not aliased to `prev`. We explicate this property as an assertion in the program.

To verify this assertion using our technique, we simply need to provide the two predicates:

```
/*@ loop_predicate prev == null, prev.val > v */
```

Using these predicates, our technique infers the loop invariant:

```
/*@ loop_invariant prev == null || !(prev.val > v) */
```

The correctness of the assertion follows from this loop invariant and from the two conditions (`curr != null` and `curr.val > v`) on the path from loop entry to `L`.

In contrast, because SLAM performs predicate abstraction for each individual statement, it requires the following two additional predicates to track the values of the two conditions.

```
    curr == null, curr.val > v
```

Since deriving predicates may still require manual intervention (both in our system and in SLAM), this need for extra predicates results in an increased burden on the programmer. In addition, by abstracting only when necessary, our approach requires many fewer theorem prover queries. In particular, using all four predicates, we require only 27 queries to verify the non-aliasing assertion whereas SLAM requires 263 queries.

7.2 Selection sort

The next example illustrates that our method works on programs with nested loops and is able to infer complex loop invariants. The method `sort`, shown in Figure 6, takes an array of integers `a` and sorts it in place. The method contains two nested loops. The outer loop uses the integer variable `i` to iterate over the elements of `a`. The inner loop finds the smallest element of `a` at or after the index `i`, which is then swapped with the element at `i`.

The outer loop ensures that the `i`-th prefix of `a` is sorted. Based on the annotations

```
/*@ skolem_constant int x, y;
    loop_predicate i >= 0, 0 <= x, x < i, x < y,
                   y < a.length, a[x] <= a[y]; */
```

our technique infers the required invariants for the outer loop:

```
/*@ loop_invariant (\forall int x, y;
                   0 <= x && x < i && x < y && y < a.length
               ==> a[x] <= a[y]);
    loop_invariant 0 <= i; */
```

The second loop invariant is needed to prove that there are no array bounds violations in the loop.

The inner loop computes in k and w the index and the value respectively of the least element with an index in the range $[i, j)$. Based on the annotations

```
/*@ skolem_constant int z;
    loop_predicate w == a[k], i <= z, z < j, w <= a[z],
                   i <= k, k < j, j <= a.length; */
```

our technique infers the required invariants for the inner loop:

```
/*@ loop_invariant w == a[k];
    loop_invariant (\forall int z; i <= z && z < j
                                 ==> w <= a[z]);
    loop_invariant i <= k;
    loop_invariant k < j;
    loop_invariant j <= a.length; */
```

7.3 The Java front-end toolkit

The examples presented so far illustrate various aspects of our technique, but are too small to provide compelling evidence that our technique is practical on large, realistic programs. Therefore, we next consider the application of our technique to a significantly larger program. The benchmark we use is the front-end to ESC/Java called Javafe, which consists of 44,388 lines of code, with 2418 routine definitions and 520 loops. This code was already annotated with appropriate ESC/Java lightweight method specifications, but did not include any loop invariants. Instead, the code was checked by first unrolling each loop some small finite number of times, typically 1 or 2. Although this unrolling technique is clearly unsound, it significantly reduces the annotation burden of using ESC/Java. Using this unsound loop analysis, ESC/Java can verify all of the routines in the benchmark.

In comparison, when using the sound loop desugaring of Section 3 (without providing or inferring any loop invariants) ESC/Java is unable to verify 326 of the 2418 routines in this program, including most of the routines containing non-trivial loops. Manually providing suitable invariants for all of these loops is a daunting task. We have been unwilling to invest this effort on our own code base, and do not expect software engineers to do so either.

However, using the techniques presented in this paper, we have extended ESC/Java to perform a sound analysis of loops, without significantly increasing the annotation burden. To achieve this sound loop analysis, we did not provide any additional annotations, such as loop invariants or loop predicates. Instead, we first used the heuristics of Section 6, which suggested an average of 3.6 predicates for each loop although it suggested many more predicates for complex loops. We subsequently used predicate abstraction to infer universally-quantified loop invariants over these predicates, which produced an average of 2.6 invariants per loop although again producing many more invariants for complex loops. Using these inferred invariants, ESC/Java verifies almost all (98.7%) of the routines in the program.

An inspection of the remaining 31 failing routines showed that they all require subtle loop invariants. For these routines, the appropriate invariants or component predicates must still be provided manually. However, the techniques of this paper reduce the number of routines for which such manual annotation is necessary by order of magnitude, from 326 to 31. In addition, our technique decreases the cost of such manual annotation by reducing the problem of specifying correct loop invariants to the problem of writing possibly useful predicates.

We believe the overhead of providing these annotations is justified by the increased rigor of the sound loop analysis, which is capable of detecting additional errors. In particular, during our inspection of the 31 failing routines, we uncovered several routines that could actually crash due to array bounds violations. The driving goal of ESC/Java is to identify such possible defects; however, ESC/Java's initial, unsound treatment of loops caused it to originally miss these defects.

7.4 Experiments

The results presented above have focused on the effectiveness of our invariant inference technique; we next discuss the computational cost of this technique. We analyzed the Javafe benchmark using three different but equivalently-precise predicate abstraction algorithms: our new algorithm (FQ), and the two existing algorithms of Das, Dill and Park (DDP), and Saïdi and Shankar (SS).

Figure 7 compares these algorithms and describes how the performance of each algorithm varies according to the number of predicates in each loop. For loops with six or fewer predicates, all algorithms perform comparably well. However, as the number of predicates increases, we begin to see significant differences in the behavior of the three algorithms. The data point for 14 predicates is somewhat of an outlier. There is only one loop in Javafe for which our heuristics generate 14 predicates, and all algorithms perform well on this loop because its invariants are particularly simple. Apart from this loop, the data suggests that our new algorithm scales better than earlier algorithms to large numbers of predicates. This scalability is an important characteristic of our predicate abstraction algorithm. In particular, it gives us the freedom to introduce additional

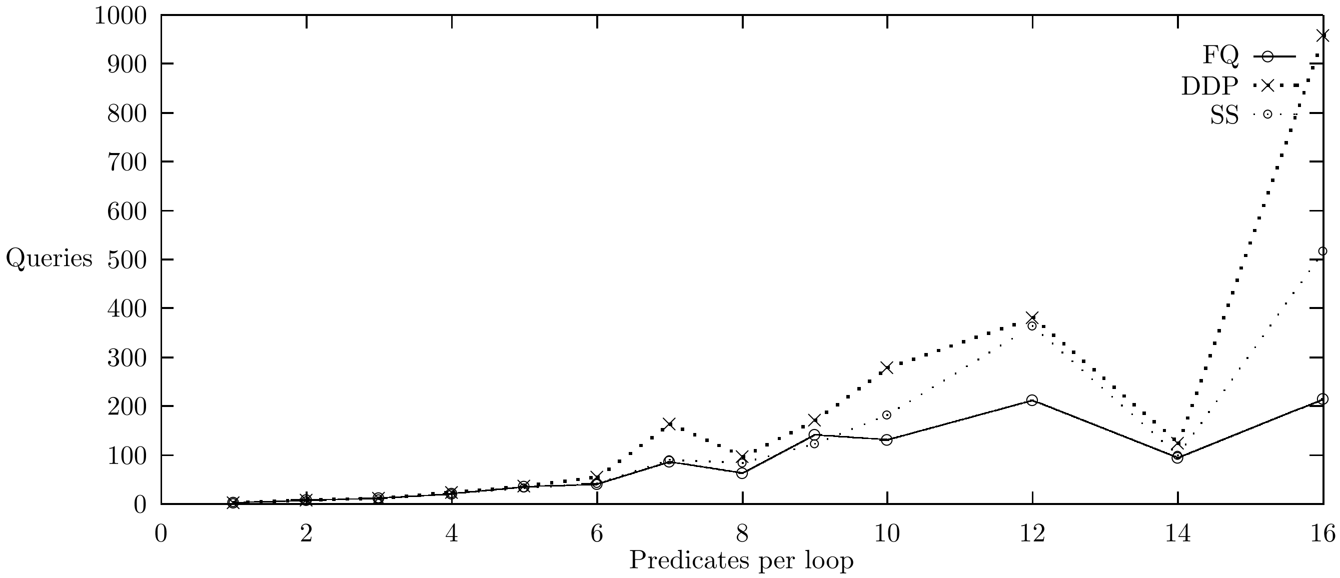

Figure 7: Comparison of the three predicate abstraction algorithms (FQ, DDP, and SS) on Javafe

Benchmark	Number of predicates	Queries		
		FQ	DDP	SS
`partition`	4	27	28	41
`sort` (*outer*)	6	44	54	111
`sort` (*inner*)	7	37	32	40
`find`	8	111	110	129
`create`	15	358	1191	2012

Figure 8: Comparison of the three predicate abstraction algorithms (FQ, DDP, and SS) on the micro-benchmarks

predicate-generating heuristics, should such additional heuristics become necessary when examining other programs.

We performed these experiments using a 667MHz EV7 Alpha processor. Over the entire Javafe benchmark (44,380 LOC), our algorithm performed 11901 queries, which required roughly 40 minutes of theorem proving time (an average of 0.2 seconds per queries). The entire loop invariant inference process took a little under an hour, proceeding at a rate of roughly 13 lines per second. We expect further progress in automatic theorem proving to significantly improve this rate.

As a second experiment, we also applied our method to to verify a Java model of the procedure `create` from the file system Frangipani [TML97]. This procedure contains a loop that ensures that the newly-created file does not duplicate the name of an existing file in the same directory. Due to the sophistication of the file system's data structures, the verification of this loop requires 15 moderately complex predicates. The performance of the the three predicate abstraction algorithms on this benchmark, and on the three other micro-benchmarks (`find`, `partition`, and the two loops in `sort`) is shown in Figure 8.

8 Conclusions

The ESC/Java project has shown that extended static checking can find a variety of errors in large programs. But the perceived burden of annotating programs has been a big hurdle to the use of static checking in program development. Although programmers are willing to document exported methods, they are not willing to specify every method and write invariants for every loop in the program.

This paper helps make extended static checking more acceptable to programmers by reducing the annotation burden of loop invariants. We present heuristics for generating appropriate predicates for each loop. Using these predicates, our loop invariant inference algorithm infers (universally-quantified) loop invariants. Our algorithm abstracts only when necessary, thus reducing the number of predicates required. Finally, our invariant inference algorithm exploits a new predicate abstraction algorithm.

The combination of these techniques reduces the annotation burden of loop invariants by an order of magnitude. In particular, we can now perform an automatic yet sound analysis of the loops in over 90% of the 396 loop-containing routines in Javafe. For the remaining 31 routines that require predicates not generated by our heuristics, our approach still reduces the problem of specifying correct loop invariants to the simpler problem of writing possibly-useful loop predicates.

Acknowledgments

We gratefully acknowledge the contributions of Chandu Thekkath, who helped us to model the method `create` of Frangipani in Java, and the ESC/Java team whose work provided the infrastructure that enabled us to implement and evaluate our ideas.

References

[BBM97] N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 325–335. Springer-Verlag, 1996.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BvW98] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages.* ACM Press, 1977.

[CGJ+00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E.A. Emerson and A.P. Sistla, editors, *CAV 2000: Computer Aided Verification*, Lecture Notes in Computer Science 1855, pages 154–169. Springer-Verlag, 2000.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual Symposium on Principles of Programming Languages*, pages 84–96. ACM Press, 1978.

[DDP99] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In N. Halbwachs and D. Peled, editors, *CAV 99: Computer Aided Verification*, Lecture Notes in Computer Science 1633, pages 160–171. Springer-Verlag, 1999.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DLNS98] D. L. Detlefs, K. R. M. Leino, C. G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[FS01] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, 1997.

[GW74] I. Greif and R. Waldinger. A more mechanical heuristic approach to program verification. In *Proceedings of the International Symposium on Programming*, pages 83–90, 1974.

[GW75] S.M. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, SE-1(1):68–75, 1975.

[KM76] S.M. Katz and Z. Manna. A logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.

[LSS99] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[Nel81] C. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.

[Nel89] C. G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[SI77] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Proceedings of the 4th Annual Symposium on Principles of Programming Languages*, pages 132–143. ACM Press, 1977.

[SS99] S. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *CAV 99: Computer Aided Verification*, Lecture Notes in Computer Science 1633, pages 443–454. Springer-Verlag, 1999.

[TML97] C.A. Thekkath, T. Mann, and E.K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[Weg74] B. Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–112, 1974.