Snugglebug: A Powerful Approach To Weakest Preconditions

Satish Chandra   Stephen J. Fink   Manu Sridharan

IBM T. J. Watson Research Center

{satishchandra,sjfink,msridhar}@us.ibm.com

Abstract

Symbolic analysis shows promise as a foundation for bug-finding, specification inference, verification, and test generation. This paper addresses demand-driven symbolic analysis for object-oriented programs and frameworks. Many such codes comprise large, partial programs with highly dynamic behaviors—polymorphism, reflection, and so on—posing significant scalability challenges for any static analysis.

We present an approach based on interprocedural backwards propagation of weakest preconditions. We present several novel techniques to improve the efficiency of such analysis. First, we present *directed call graph construction*, where call graph construction and symbolic analysis are interleaved. With this technique, call graph construction is guided by constraints discovered during symbolic analysis, obviating the need for exhaustively exploring a large, conservative call graph. Second, we describe *generalization*, a technique that greatly increases the reusability of procedure summaries computed during interprocedural analysis. Instead of tabulating how a procedure transforms a symbolic state in its entirety, our technique tabulates how the procedure transforms only the pertinent portion of the symbolic state. Additionally, we show how integrating an inexpensive, custom logic simplifier with weakest precondition computation dramatically improves performance.

We have implemented the analysis in a tool called SNUGGLEBUG and evaluated it as a bug-report feasibility checker. Our results show that the algorithmic techniques were critical for successfully analyzing large Java applications.

Categories and Subject Descriptors   D.2.4 [*Software/Program Verification*]: Validation; D.2.5 [*Testing and Debugging*]: Symbolic execution

General Terms   Algorithms, Languages, Verification

Keywords   Interprocedural analysis, symbolic analysis, weakest preconditions

1. Introduction

We consider the problem of finding a precondition ϕ that necessarily drives a program from a particular entrypoint m to a particular goal state g. A general solution to this problem would have numerous applications in tools for software engineering, such as:

- **Specification discovery and API hardening** Here, g could represent some behavior of a library, and a discovered precondition would illustrate how to make such behavior occur. For example, g might represent that the library throws a particular exception at a particular line of code. Presented with a precondition for an exceptional exit, a library developer might either change the code to avoid the exception, or add the precondition to the documentation.

- **Bug validation** Such analysis could reduce the impact of false positives from a bug-finding tool, by treating each bug report as a goal state and searching for sufficient preconditions. When a tool finds a sufficient precondition for a bug report, said report would be considered "validated", and hence deserve higher priority.

- **Test case generation** Given a precondition ϕ for method foo, one may wish to construct a test case that executes foo in a state satisfying ϕ (*e.g.*, as a debugging aid). Suppose we construct a "universal driver" program that can execute candidate sequences of method calls, which embody a space of possible tests. If we can force the universal driver to the goal state of ϕ at the entry of foo, a tool could output the corresponding method sequence as the desired test.

In this paper, we desire a *sound* solution to the goal-reachability problem, one that models the exact semantics of all statements, including interprocedural flow and exceptional conditions. When the analysis finds a precondition ϕ for g, we insist that the analysis *guarantee* that any state which satisfies ϕ must necessarily drive program execution to g. No other exceptions will be thrown before reaching g. A sound analysis must necessarily perform full interprocedural analysis, since the analysis cannot optimistically assume that certain method calls have no side effects or even return normally.

We propose a solution to this goal-reachability problem based on *backward symbolic analysis.* In principle, such an analysis computes *weakest preconditions* [12] over each control-flow path, going backwards from the goal statement to the entrypoint. If the computed precondition ϕ for any path r is satisfiable, then a satisfying assignment for ϕ gives inputs that would force execution along r to the goal.

Backward symbolic analysis suits our problem for a number of reasons. The analysis is sound and models concrete semantics (*i.e.*, no abstraction), and hence it outputs no false positives. It is also *demand driven*; by design, it explores only states that are relevant to reaching the goal, unlike symbolic execution [23] and most testing-based techniques (*e.g.*, [17]). Finally, in contrast to testing, backward symbolic analysis does not require a complete program; it can analyze libraries without an execution environment and client code.

Backward symbolic analysis with concrete semantics is necessarily incomplete. It may fail to find a suitable precondition, even if one exists, in the presence of loops and recursion. This is not

an immediate disqualification of its utility: our experiments show that for goals arising in realistic programs, a systematic search of executions even through programs containing loops and recursive procedures *often* succeeds in producing a suitable precondition.

Challenges Real-world programs present many challenges for weakest precondition (*wp*) analysis. The first problem arises from the sheer scale of large programs. Even in loop-free programs, symbolic analysis faces an exponential explosion due to the number of distinct paths through the program. In straight-line code alone, handling language features like aliasing and type tests can require disjunctions, another source of state explosion.

Procedure calls both exacerbate these difficulties and introduce entirely new challenges, especially for large object-oriented libraries and frameworks. For object-oriented programs, performing interprocedural analysis requires determining the possible targets of virtual method calls. Unfortunately, standard call graph construction algorithms [18] face myriad difficulties disambiguating virtual calls in real-world libraries, due to the scale of the programs, unknown aliasing that clients might establish, and dynamic language features like reflection. Analyzing all possible virtual call targets as computed by simpler techniques (*e.g.*, using the type system) dramatically reduces analysis scalability.

Even if call graph difficulties were resolved, the standard challenges of interprocedural analysis remain. The best-known previous work in this area, ESC/Java [15], performed only intraprocedural analysis, requiring programmers to provide appropriate specifications for called methods. However, the concomitant annotation burden of this approach can be heavy even with tool assistance [14]. The simplest approach to automatic interprocedural analysis is to inline callees. However, inlining fails to terminate with recursion, and furthermore often leads to an exponential explosion in program size; moreover, it sacrifices any possibility of reuse of analysis of a procedure. An alternative solution, long investigated for interprocedural dataflow analysis [29], works by computing procedure summaries automatically and reusing them to reduce redundant computation. The challenge in this approach, when applied to symbolic analysis, is to efficiently compute summaries that are general enough to reuse frequently.

Contributions We present an approach to interprocedural weakest precondition computation based on Sharir and Pnueli's functional interprocedural analysis framework [29]. Our key contributions lie in the following techniques, which enable analysis of real-world object-oriented programs:

1. *Directed Call Graph Construction* We show an iterative algorithm that interleaves symbolic analysis with call graph construction. The call graph grows in stages, driven by feedback from symbolic analysis, in an attempt to explore only callees consistent with the goal. Furthermore, the technique requires no whole-program analysis beyond class hierarchy construction.

2. *Generalization* We describe a technique to enhance summary reuse via computation of generalized procedure summaries, in the context of the Reps-Horwitz-Sagiv (RHS) tabulation-based analysis framework [26].

3. We show how integrating an inexpensive, custom logic simplifier with weakest precondition computation dramatically improves performance.

We have implemented this approach in a tool called SNUGGLEBUG and evaluated it on challenging bug-validation tasks from large framework-oriented Java applications, including the open-source Tomcat web server and the Eclipse IDE. SNUGGLEBUG succeeded in establishing concrete preconditions for 29 out of 38 feasible goals it considered, each within a half-hour limit. To our knowl-

edge, this paper is the first to successfully apply demand-driven interprocedural symbolic analysis to programs of this scale.

Our evaluation showed that the techniques listed above were critical for successful analysis of these applications. Precomputed call graphs (without directed call graph construction) often encompassed thousands of methods, can overwhelm symbolic analysis, whereas the directed call graph construction needed at most 93 methods. Generalization improved the frequency of summary-edge reuse to 87% from 5%, resulting in a factor of two bottom-line speedup. Finally, the integrated custom simplifier improved performance by roughly a factor of ten.

The remainder of this paper proceeds as follows. Section 2 gives an overview of our techniques, and Section 3 describes our core analysis architecture. Section 4 presents directed call graph construction, and Section 5 shows generalization. Section 6 presents other details of our system relevant to performance. Section 7 shows experimental results, Section 8 discusses related work, and Section 9 concludes.

2. Overview

Here, we give an overview of the key techniques employed in SNUGGLEBUG, using the examples in Figures 1 and 2. For Figure 1, we wish to discover a precondition for the public `entrypoint` method that will force program execution to line 38, which throws an exception. This precondition could be useful either to find bugs or as documentation. In this case, we shall discover that line 38 is reached if `entrypoint` is invoked with a `NewCarList` containing a `Car` whose year is not 2009.

First we give an informal taste of the weakest precondition (*wp*) propagation [12] at the core of SNUGGLEBUG. Starting backwards from the goal at line 38 of Figure 1, *wp* computes the following symbolic states, all preconditions for reaching the error:

1. Before line 38, `true`
2. Before line 37, $y \neq 2009$
3. Before line 36, $x.\text{year} \neq 2009$.
4. Before line 35, $x.\text{year} \neq 2009 \land newCarsOnly$

and so on. At each step, we apply the statement's *wp* transformer to a postcondition to arrive at a precondition.

The key challenge in analyzing Figure 1 is handling its virtual method calls. Continuing the above *wp* computation backwards in `checkValid()` requires analyzing calls to `Iterator.next()`, `Iterator.hasNext()`, and `Collection.iterator()`. These calls can have many possible targets in the application or libraries—had this code relied on the Eclipse UI subsystem, there would be at least 86 concrete implementations of `iterator()`, 135 implementations of `hasNext()`, and 157 implementations of `next()`. *Which of these methods should the analysis explore?* Answering this question accurately is a requirement for efficient analysis of this example. In this case, when line 38 is reached from `entrypoint`, a path condition constrains the receivers of these calls to the particular types `NewCarList` and `NewCarList$Itr`.

2.1 Directed Call Graph Construction

Directed call graph construction works by iteratively *skipping* analysis of certain virtual calls and then choosing call targets based on feedback from symbolic analysis. In the first pass of backward analysis of Figure 1, we skip the calls to `iterator()`, `hasNext()`, and `next()` in `checkValid()`, reaching entry of `entrypoint` with a formula that *symbolically* represents their return values and possible effects. On any path from line 38 to the entry of `entrypoint`, we have the constraint that `c` must be a subtype of `NewCarList` (due to line 28). With this constraint, the analysis next decides to expand the `iterator()` call at line 32 to include

```
 1  public class Car {
 2    int year;
 3    void setYear(int y) { this.year = y; }
 4    int getYear() { return year; }
 5  }
 6  public class NewCarList implements List<Car> {
 7    private Car[] elems = new Car[10];
 8    public final Iterator iterator() {
 9      return new Itr();
10    }
11    public Car set(int i, Car c) {
12      Car old = elems[i];
13      elems[i] = c;
14      return old;
15    }
16    private class Itr implements Iterator {
17      int cursor = 0;
18      public boolean hasNext() {
19        return cursor != elems.length;
20      }
21      public Object next() {
22        return elems[cursor++];
23      }
24    }
25    // other List methods...
26  }
27  public static void entrypoint(Collection<Car> c)
28    checkValid(c, c instanceof NewCarList);
29  }
30  private static void checkValid(Collection<Car> c,
31    boolean newCarsOnly) throws MyException {
32    Iterator<Car> it = c.iterator();
33    while (it.hasNext()) {
34      Car x = it.next();
35      if (newCarsOnly) {
36        int y = x.getYear();
37        if (y != 2009) {
38          throw new MyException(); // GOAL
39        }
40      }
41      // ... other checks.
42    }
43  }
```

Figure 1. A (contrived) motivating example

```
 1  public static void test(NewCarList l) throws MyException {
 2    Car c1 = new Car();
 3    l.set(0,c1);
 4    c1.setYear(2008); // a bad car
 5    Car c2 = new Car();
 6    l.set(1,c2);
 7    c2.setYear(2009); // a good car
 8    entrypoint(l);
 9  }
```

Figure 2. Motivating example for modular reuse.

of input-output facts for each procedure. The algorithm maintains a table of input-output summaries for each procedure, and reuses a computed summary when it propagates an input fact that yields a "hit" in the summary table.

With symbolic formulae as dataflow facts, our interprocedural analysis will "hit" in the summary table only when presented with a *syntactically identical* formula. However, even when using canonical forms to represent symbolic formulae, summary reuse based on syntactic matching often fails. We present *generalization*, a technique to compute more general symbolic summaries. Generalization lifts summaries over individual symbolic facts into summaries over more general classes of facts.

We now show informally how generalization applies to the example code in Figure 2. Here, the `test()` method invokes `entrypoint()` (from Figure 1) with a `NewCarList` that causes the previously discussed exception at line 38 in `checkValid()`. Note that `test()` contains two calls to each of `NewCarList.set()` and `Car.setYear()`; we shall show how generalization enables analyzing each of these methods once and re-using the result.

Consider the symbolic states that arise during wp analysis immediately after the call sites to `setYear()` and `set()` in Figure 2 (certain irrelevant conjuncts elided for brevity):

Method, line	Postcondition
`setYear()`, 7	$l.\texttt{elems}[0].\texttt{year} \neq 2009 \wedge l.\texttt{elems.length} > 0$
`setYear()`, 4	$l.\texttt{elems}[0].\texttt{year} \neq 2009 \wedge l.\texttt{elems.length} > 1$
`set()`, 6	$l.\texttt{elems}[0] \neq c_2 \wedge l.\texttt{elems}[0].\texttt{year} \neq 2009$
`set()`, 3	$\texttt{read}(\texttt{update}(\texttt{year}, c_1, 2008), l.\texttt{elems}[0]) \neq 2009$

The formula notation will be explained further in Section 3; for now, just notice that the two formulae for `setYear()` are syntactically different; likewise for `set()`. Since these formulae are the input keys in the summary tables for `setYear()` and `set()`,[1] the tabulation-based analysis cannot achieve any reuse for the repeated calls.

Generalization enables computation of more general *symbolic summaries* [10] within a tabulation-based interprocedural analysis by (1) extracting references to concrete locations in input formulae for procedures and (2) using the frame rule from separation logic [27] to analyze procedures only with relevant, genericized conjuncts. Generalization of the postcondition at line 7 yields:

$$l_0 \neq 2009 \wedge l_0 = \texttt{read}(\texttt{year}, l_1)$$
$$\wedge \, l_1 = l.\texttt{elems}[0] \wedge l.\texttt{elems.length} > 0$$

Compared to the corresponding formula in the table above, the reference to the `year` field has been extracted into a generic conjunct $l_0 = \texttt{read}(\texttt{year}, l_1)$ (l_0 and l_1 are fresh variables; `read` is discussed in Section 3). The analysis then reasons via the frame rule [27] that since `setYear()` may modify only `year`, only this generic conjunct is relevant when analyzing the method. Hence, the algorithm propagates only the generic conjunct $l_0 = \texttt{read}(\texttt{year}, l_1)$, to the callee, yielding the following entry in

`NewCarList.iterator()` as a target for analysis. In this manner, symbolic analysis gives crucial *feedback* to the interprocedural propagation.

Newly-added callees can influence which methods are added to the call graph in later analysis stages. For our example, after adding `NewCarList.iterator()` to the call graph, the analysis discovers that it returns an object of type `NewCarList$Itr` (line 9). This fact constrains the call graph expansion for the calls at lines 33 and line 34, forcing analysis of methods in `NewCarList$Itr` for subsequent passes. Notice how the call target selection is guided by the *latest* known symbolic constraints, in this case the concrete type of the return value from `NewCarList.iterator()`.

Directed call graph construction improves scalability because in practice, the analysis needs to only explore a small portion of an overapproximate (worst case) call graph. An upfront static analysis would have difficulty determining the right part of the call graph to explore—it is the interleaving of interprocedural symbolic analysis and call graph construction that makes this strategy work.

2.2 Generalization

Our interprocedural analysis algorithm embodies a functional interprocedural dataflow analysis [29] using an adaptation of the RHS tabulation algorithm [26]. In our case, the domain of this dataflow problem consists of symbolic formulae, where each formula represents a set of concrete program states. The tabulation algorithm builds partial procedure summaries on-the-fly, as it discovers pairs

---

[1] In the actual tables, the formulae are translated into the callee namespace.

`setYear()`'s input-output table:

$$l_0 = \texttt{read}(year, l_1) \xrightarrow{\texttt{setYear()}} l_0 = \texttt{read}(\texttt{update}(year, this, y), l_1)$$

The above represents a symbolic summary that applies to an entire class of facts, namely those that reference the `year` field. Generalization of the post-condition for the `setYear()` call at line 4 again yields the input fact $l_0 = \texttt{read}(year, l_1)$, enabling reuse of the above summary via tabulation. Similar reuse is achieved for the calls to `set()`, as we shall show in Section 5.

Generalized summaries promote reuse because methods frequently embody context-independent behaviors. SNUGGLEBUG computes generalized summaries *on demand* within the interprocedural *wp* computation, consistent with the overall demand-driven analysis approach. Section 5 gives further details on how to integrate generalized summary computation into a standard tabulation-based analysis [26].

2.3 Integrated Tabulation and Simplification

Even with directed call graph construction and generalization, straightforward propagation of weakest preconditions would not scale to the size of programs we handle. In particular, we found that an approach of constructing weakest preconditions compositionally and sending the results to an SMT solver to check decidability (as was done in previous systems [15]) was not practical for our target programs.

A further key to scalability lies in close co-operation between *wp* propagation and formula handling. At each step of *wp* propagation, SNUGGLEBUG invokes an inexpensive custom procedure to simplify formulae using language-specific theories, and it drops formulae from further propagation once proven unsatisfiable. This technique also increases the effectiveness of directed call graph construction and generalization. Section 6 describes our formula handling and other optimizations.

3. Analysis Basics

In this section, we first describe an intraprocedural *wp* calculation for Java (Section 3.1) and then extend the analysis to handle procedure calls (Section 3.2). SNUGGLEBUG extends this core analysis with directed call graph construction (Section 4) and enhancements for modular reuse (Section 5).

3.1 Intraprocedural Computation

Our intraprocedural analysis operates on a SSA register-transfer language representation with semantics close to Java bytecode. The first column of Table 1 shows some of the statements in the language; their semantics follow those of Java. The *assume* statement is a no-op if its condition is true and hangs otherwise. Following a conditional branch with condition c, the taken branch jumps to $assume(c)$ and the not taken branch to $assume(!c)$. Note that many statements can throw implicit exceptions corresponding to built-in Java safety conditions. We abbreviate null pointer, class cast, and array index out of bounds exceptions as NPE, CCE, and OOB respectively.

The analysis operates on a control-flow graph (CFG) built over this representation, where each basic block has at most one statement.[2] Each CFG has a unique *Entry* and unique *Exit* node. Each block has distinct outgoing edges corresponding to normal execution and different cases of exceptional execution. Exceptional edges from a potentially excepting statement go to either catch blocks or the exit node.

SNUGGLEBUG represents symbolic states (the domain of the analysis) as quantifier-free formulae in first-order logic with equality. Table 2 informally presents some of the vocabulary of this logic

Functions and Constants	
T_i	type constants (one per concrete type)
M_i	method constants (one per concrete method)
F_i	field constants (one per declared field)
sig_i	constant corresponding to a method signature
$\texttt{read}(f,v)$	$f(v)$, where f is $Val \rightarrow Val$
	(a relational model of some declared field)
$\texttt{update}(f,v,w)$	functional update of f, i.e. $f[v \mapsto w]$
$\texttt{aread}(a,v,i)$	$a(v,i)$, where a is $Val \times Index \rightarrow Val$
$\texttt{aupdate}(a,w,i,v)$	functional update of a, i.e. $a[(w,i) \mapsto v]$
$\texttt{typeOf}(v)$	the type of object to which v points
$\texttt{dispatch}(t,sig)$	method to which signature sig
	will dispatch to on receiver of type t
$\texttt{subType}(t_1,t_2)$	true iff type t_1 is subtype of type t_2 in Java
Axioms	
$this \neq \texttt{null}$	
$\forall f. \forall v. \forall w. \texttt{read}(\texttt{update}(f,v,w),v) = w$	
$\forall f. \forall v. \forall w. \forall u. u \neq v \Rightarrow \texttt{read}(\texttt{update}(f,v,w),u) = \texttt{read}(f,u)$	
$\texttt{subType}(T_1, T_2) \Leftrightarrow T_1$ is a subtype of T_2 in Java	
$\texttt{dispatch}(T_1, sig_{A.m}) = M_{B.m} \Leftrightarrow$	
$\quad \forall x.(x.\texttt{class} = T_1 \Rightarrow x.\texttt{m}()$ dispatches to target method $\texttt{B.m}())$	
$\forall x. \texttt{read}(length, x) \geq 0$	

Table 2. Some representative symbols and axioms in our theory for Java programs.

and shows some representative axioms.[3] In addition to primitive values and pointers, the vocabulary expresses relations between types, methods, fields, and method signatures. The logic models Java fields as relations manipulated with *read* and *update* from the theory of arrays [25]. Java arrays (which are heap-allocated) are modeled with two-dimensional arrays (from the theory), indexed by a base pointer and an array index. The bottom half of Table 2 shows some representative axioms, such as the standard axioms that define the theory of arrays. Additionally, the table shows axioms based on the type hierarchy of the program. The last axiom ensures that the `length` field of arrays is non-negative.

Table 1 defines the weakest precondition transformers for several statements. The notation $\phi[t_2/t_1]$ means ϕ with all syntactic occurrences of t_1 replaced by t_2. Technically, the *wp* transformer occurs on an outgoing edge from a basic block (recall that each basic block has at most one statement). For some statements, *wp* must take into account whether the CFG edge represents normal or exceptional control flow, as indicated in the second column of Table 1. Note that the *wp* transformer for calls in Table 1 only handles intraprocedural semantics (*i.e.*, reasoning about non-nullness of receivers and virtual dispatch); Section 3.2 discusses interprocedural analysis.

Figure 3 gives pseudocode for an iterative computation of intraprocedural weakest pre-conditions. The analysis computes a set (D) of symbolic states at each program point. After computing $wp(s, \phi_{post})$ for a statement s and formula ϕ_{post}, the algorithm (i) invokes a simplifier on the result and (ii) merges the simplified result with the facts already present before s (line 8). We use a lightweight but highly effective simplifier for (i), described in Section 6. For (ii), we perform ad hoc checks to optimize away certain patterns of redundancy: for example, $merge(\phi \wedge a, \phi \wedge !a)$ simplifies to ϕ; it is only necessary to propagate ϕ further. In the presence of loops, INTRAWP may not terminate, but in practice, we only need to run it until a suitable $D(entry)$ is obtained, even if it is not the *weakest* such condition. (Section 6 contains more details on handling of loops.)

Example Consider the method `setYear` in Figure 1. Suppose we wish to find the precondition for the predicate `x.year == 2008`

---

[2] We elide some straightforward details involving SSA ϕ statements.

[3] Though the axioms include quantifiers, they never arise in the symbolic state representation.

statement	edge condition	$wp(statement, \phi)$
$v = w$		$\phi[w/v]$
$v = v_1 \ op \ v_2$		$\phi[(v_1 \ op \ v_2)/v]$
$v = w.f$	normal successor	$(w \neq \texttt{null}) \wedge \phi[\texttt{read}(f, w)/v]$
	NPE successor	$(w = \texttt{null}) \wedge \phi[fresh(\texttt{NPE})/exc]$
$v.f = w$	normal successor	$(v \neq \texttt{null}) \wedge \phi[\texttt{update}(f, v, w)/f]$
	NPE successor	$(v = \texttt{null}) \wedge \phi[fresh(\texttt{NPE})/exc]$
$v = w[i]$	normal successor	$(w \neq \texttt{null}) \wedge (i < \texttt{read}(length, w) \wedge i \geq 0) \wedge \phi[\texttt{aread}(a, w, i)/v]$
	NPE successor	$(w = \texttt{null}) \wedge \phi[fresh(\texttt{NPE})/exc]$
	OOB successor	$(w \neq \texttt{null}) \wedge (i < 0 \vee i \geq \texttt{read}(length, w)) \wedge \phi[fresh(\texttt{OOB})/exc]$
$w[i] = v$	normal successor	$(w \neq \texttt{null}) \wedge (i < \texttt{read}(length, w) \wedge i \geq 0) \wedge \phi[\texttt{aupdate}(a, w, i, v)/a]$
	NPE successor	$(w = \texttt{null}) \wedge \phi[fresh(\texttt{NPE})/exc]$
	OOB successor	$(w \neq \texttt{null}) \wedge (i < 0 \vee i \geq \texttt{read}(length, w)) \wedge \phi[fresh(\texttt{OOB})/exc]$
$v = new \ T$		$\phi[fresh(T)/v]$
$assume \ c$		$\phi \wedge c$
$v_1 = (T) \ v_2$	CCE successor	$v_2 \neq \texttt{null} \wedge \neg(\texttt{subType}(\texttt{typeOf}(v_2), T)) \wedge \phi[fresh(\texttt{CCE})/exc]$
	normal successor	$(v_2 = \texttt{null} \vee \texttt{subType}(\texttt{typeOf}(v_2), T)) \wedge \phi[v_2/v_1]$
$return \ v$		$\phi[\texttt{null}/exc][v/ret]$
$w = v.m()$	normal successor, callee $meth$	$meth = \texttt{dispatch}(\texttt{typeOf}(v), m()) \wedge v \neq \texttt{null}$
	NPE successor	$(v = \texttt{null}) \wedge \phi[fresh(\texttt{NPE})/exc]$

Table 1. Specification of wp for representative Java statements. v, w and c variables represent symbolic registers in the input language, and corresponding free variables in the logic; they can hold values of either primitive types (integers, reals) or pointers. exc is a special variable that holds a pointer to an exception that has been raised but not caught, and ret represents the return value. $fresh(T)$ returns a fresh value v from the domain of pointers, such that $\texttt{typeOf}(v) = T$.

INTRAWP(CFG, $postcondition$)

```
 1   var D: Statement → {Formula}
 2   var worklist: stack of (Statement, Formula)
 3   ∀s ∈ Statement , D(s) ← ∅
 4   worklist ← {(Exit, postcondition)}
 5   while worklist is not empty
 6       do (s', φ_post) ← get from worklist
 7          for each s such that (s, s') is an edge in CFG
 8              do φ_pre ← merge(D(s), simplify(wp(s, φ_post)))
 9                 if φ_pre ≠ FALSE
10                     then D(s) ← D(s) ∪ φ_pre
11                          add (s, φ_pre) to worklist
12   return D(Entry)
```

Figure 3. Computation of intraprocedural weakest pre-conditions.

at the normal exit of `setYear`, where x is some global variable. The symbolic state representing this predicate and normal execution of `setYear` is $exc = \texttt{null} \wedge \texttt{read}(year, x) = 2008$ at the exit of `setYear`. Traversing `setYear` backwards, the analysis first encounters a return statement (not shown in code). Applying the wp transformer for the return statement substitutes `null` for exc, giving $\texttt{read}(year, x) = 2008$. Applying the wp transformer for the putfield statement at line 3, we get $this \neq \texttt{null} \wedge \texttt{read}(\texttt{update}(year, this, y), x) = 2008$. Before propagating this formula, the simplifier uses the axiom $this \neq \texttt{null}$ to obtain the simpler formula $\texttt{read}(\texttt{update}(year, this, y), x) = 2008$. If later in the analysis, some path condition ensures that $this = x$, then the simplifier will further simplify, based on the theory of arrays, resulting in $y = 2008$.

3.2 Interprocedural Computation

We next describe interprocedural analysis assuming some oracle has provided a call graph; Section 4 discusses our directed call graph construction.

Our analysis handles procedure calls in a context-sensitive manner, *i.e.*, we only consider *realizable* interprocedural paths. Context sensitivity is accomplished through a functional approach [29] based on the Reps-Horwitz-Sagiv (RHS) tabulation algorithm [26],

enhanced to handle merge functions and combination of local and non-local flows at return sites.

The analysis operates over an interprocedural control flow graph (ICFG), consisting of CFGs linked via edges from call sites to and from the *Entry* and *Exit* nodes in corresponding callee CFGs. A single worklist holds the pending work (symbolic states to propagate), *i.e.*, the algorithm does not completely analyze a callee before continuing work in the caller. The global worklist effectively manages instances of INTRAWP as co-routines. Analogously to the problem with loops in INTRAWP, the procedure may not terminate in the presence of recursion.

Propagation of a formula to and from a callee works as follows. Suppose a formula ϕ reaches a call site $w = v.m()$, and assume for there exists one possible callee $A.m$. (For multiple possible callees, the procedure is simply iterated over each callee.) The analysis first projects ϕ into $A.m$'s namespace—substituting formals for actuals, and so on—and propagates the result ϕ_{post} to $A.m$'s *Exit*. This symbolic state then propagates through $A.m$ via INTRAWP, processing any further calls recursively. Whenever as a result of this propagation, a formula ϕ_{pre} reaches $A.m$'s *Entry*, the solver records a *summary edge* $\phi_{post} \xrightarrow{A.m()} \phi_{pre}$, indicating that ϕ_{pre} is a sufficient precondition to ensure reaching ϕ_{post} at the *Exit*. Note that for a single ϕ_{post}, the solver may discover many sufficient preconditions as it explores more paths. Finally, the solver applies the summary at the call site by projecting ϕ_{pre} to the caller's namespace and conjoining it with $wp(w = v.m())$, described in Table 1. Note that the analysis caches summary edges and *reuses* them when identical formulae propagate to an *Exit* node [26]. Section 5 describes techniques to increase the effectiveness of this reuse.

4. Directed Call Graph Construction

The previous section addressed interprocedural analysis, but it neglected to address the central challenge discussed in Section 2.1: *What call graph should we use, when faced with high degrees of polymorphism?* This section presents a solution called *directed call graph construction*, which uses feedback from symbolic analysis to choose the call graph.

INTERWPDEMAND($ICFG$, $postcondition$)

1   var F: {$Formula$}
2   $F \leftarrow$ INTERWP($ICFG$, $postcondition$)
3   $F' \leftarrow \{f \in F \mid f$ has no skolems $\wedge f$ is satisfiable $\}$
4   **if** $F' \neq \emptyset$
5   **then return** F'
6   **else for** each $t \in F$
7   **do** choose σ_{method} from t
8   choose $m' \notin targets(\sigma_{method}, ICFG)$ s.t.
               $t \wedge \sigma_{method} = m'$ satisfiable
9   **if** no such m'
10   **then continue**
11         $ICFG_{new} \leftarrow ICFG$ with m' as
               possible target at $site(\sigma_{method})$
12   **return** INTERWPDEMAND($ICFG_{new}$,
               $postcondition$)
       ▷ Failed to expand call graph
13   **return** $\emptyset$

Figure 4. Pseudocode for directed call graph construction. $site(\sigma)$ denotes the call site represented by a skolem constant, $targets(\sigma, ICFG)$ gives the set of methods in the current ICFG for $site(\sigma)$.

Directed construction requires analyzing the program in stages. The first stage performs symbolic analysis while skipping over all method calls. If this analysis finds a satisfiable precondition through a path that does not have any method calls, the computation terminates, having found a call-free feasible path that reaches the goal. Otherwise, the algorithm expands the call graph, adding a callee at some call site. The key insight is that constraints from symbolic analysis guide the choice of call site and target.

We skip method calls by modifying the intraprocedural wp computation for calls from Table 1. We introduce *skolem* constants—essentially existentially quantified variables—to represent the constraints induced by a skipped method. For a call $w = v.m()$, we use four types of fresh skolem constants:

1. σ_{ret} represents the undetermined return value assigned to w.

2. σ_{method} represents the undetermined target of the method dispatch; constraints on this variable guide selection of a target when expanding the call.

3. For each field f referenced in the post-condition of the call, σ_f captures the undetermined side-effects of the method on f. We introduce an uninterpreted function mod where $\text{mod}(f, \sigma_f)$ is an array term (in the theory of arrays; see Section 3.1) that represents the updates to the relation f performed by the callee. Java arrays are handled similarly (details elided).

4. σ_{exc} represents the undetermined exception value generated during the execution of the method.

The modified wp for skipped calls follows; we show only the non-exceptional case:

$$
\begin{aligned}
wp(w = v.m(), \phi) \;=\; &\sigma_{method} = \text{dispatch}(\text{typeOf}(v), m()) \\
\wedge\; &v \neq \text{null} \\
\wedge\; &\phi[\sigma_{ret}/w][\sigma_{exc}/exc][\text{mod}(f, \sigma_f)/f]^{(*)} \\
&((*) \text{ for each field and array})
\end{aligned}
$$

Given the modified wp, Figure 4 gives pseudocode for directed call graph construction. We assume a procedure INTERWP that computes the interprocedural symbolic analysis described in Section 3.2: given an ICFG and post-condition, it returns a set of satisfiable preconditions at entry. The **for** loop from lines 6 to 12 attempts to expand the call graph. Note the satisfiability check at line 8, showing how symbolic constraints influence the choice of call targets. Finally, note that the analysis allows for expanding multiple

targets at a call site. This functionality is needed not only for calls with multiple possible targets, but also for cases when a callee is feasible according to constraints over skolem constants, but has behavior incompatible with the post-condition (*e.g.*, if we need a non-null return value and the expanded callee always returns null).

As in Figure 4, SNUGGLEBUG currently computes wp from scratch in each analysis phase. Reuse of work from previous phases could yield a large performance benefit, but bounded analysis and skolem constants make such reuse non-trivial. We plan to investigate reuse across phases further in future work.

4.1 Directed Call Graph Construction Example

Here we illustrate in detail how directed call graph construction runs on the example of Figure 1, as was discussed at a high level in Section 2.1. Recall that the goal is to find a concrete execution from the beginning of entrypoint to line 38. We focus on the loop-free backwards path π going through lines 37, 36, 35, 34, 33, 32, and 28. The starting formula that we propagate backwards from line 38 is simply true, *i.e.*, the line was executed.

Phase 1 In the first phase, the ICFG contains methods entrypoint(), checkValid() and Car.getYear().[4] During wp propagation along path π, the flow functions introduce skolem constants for the method calls iterator(), hasNext(), and next(), which do not appear in the initial ICFG and so are skipped:

- For next, $\sigma_{method,n}$, $\sigma_{exc,n}$, $\sigma_{ret,n}$, and $\sigma_{year,n}$
- For hasNext, $\sigma_{method,h}$, $\sigma_{exc,h}$, $\sigma_{ret,h}$, and $\sigma_{year,h}$
- For iterator, $\sigma_{method,i}$, $\sigma_{exc,i}$, $\sigma_{ret,i}$, and $\sigma_{year,i}$

We omit *exc* and the σ_{exc} variables from our discussion, as they are not relevant in this example.

The formula that reaches the entry of entrypoint follows, applying the appropriate flow functions from Table 1 for each statement in the path:

$$
\begin{aligned}
&\sigma_{method,n} = \text{dispatch}(\text{typeOf}(\sigma_{ret,i}), \text{next}()) \\
\wedge\; &\sigma_{method,h} = \text{dispatch}(\text{typeOf}(\sigma_{ret,i}), \text{hasNext}()) \\
\wedge\; &\sigma_{method,i} = \text{dispatch}(\text{typeOf}(c), \text{iterator}()) \\
\wedge\; &\text{read}(\text{mod}(\text{mod}(\text{mod}(year, \sigma_{year,i}), \sigma_{year,h}), \sigma_{year,n}), \sigma_{ret,n}) \\
&= 2009 \\
\wedge\; &\sigma_{ret,h} = \text{true} \\
\wedge\; &c \neq \text{null} \wedge \sigma_{ret,n} \neq \text{null} \wedge \sigma_{ret,i} \neq \text{null} \\
\wedge\; &\text{subType}(\text{typeOf}(\sigma_{ret,n}), \text{Car}) \\
\wedge\; &\text{subType}(\text{typeOf}(\sigma_{ret,i}), \text{Iterator}) \\
\wedge\; &\text{subType}(\text{typeOf}(c), \text{NewCarList})
\end{aligned}
$$

The read term arises from analyzing Car.getYear(), and the nested mod terms compositionally indicate the possible side effects of skipped methods on contents of the year field.

The preceding formula is satisfiable. However, it contains skolem constants, which indicate that the path which generates this formula skipped over some calls. Hence, INTERWPDEMAND must expand the call graph, trying to find a path with no skipped calls.

Suppose it selects to expand the call to iterator (line 32) next. The type constraint $\text{subType}(\text{typeOf}(c), \text{NewCarList})$ indicates that c must be of type NewCarList. (The constraint arose from the instanceof check at line 28.) Hence, INTERWPDEMAND concludes that $\sigma_{method,i} = \text{NewCarList.iterator}()$, expands the call graph accordingly, and recurses.

Phase 2 INTERWPDEMAND performs symbolic analysis over the expanded call graph. This time the following symbolic state

---

[4] For expository purposes, we assume the expansion of the monomorphic call to Car.getYear has already occurred.

reaches entry, indicating two skipped method calls on the path:

$$\sigma_{method,n} = \texttt{dispatch}(\texttt{NewCarList\$Itr}, \texttt{next}())$$
$$\wedge \quad \sigma_{method,h} = \texttt{dispatch}(\texttt{NewCarList\$Itr}, \texttt{hasNext}())$$
$$\wedge \quad \texttt{read}(\texttt{mod}(\texttt{mod}(\texttt{year}, \sigma_{year,h}), \sigma_{year,n}), \sigma_{ret,n}) = 2009$$
$$\wedge \quad \sigma_{ret,h} = \texttt{true} \wedge c \neq \texttt{null} \wedge \sigma_{ret,n} \neq \texttt{null}$$
$$\wedge \quad \texttt{subType}(\texttt{typeOf}(\sigma_{ret,n}), \texttt{Car})$$
$$\wedge \quad \texttt{subType}(\texttt{typeOf}(c), \texttt{NewCarList})$$

Note that since we analyze `NewCarList.iterator`, the concrete type `NewCarList$Itr` returned by the method now appears in the `dispatch` constraints. Continuing, we successively add targets `next()` and `hasNext()` in the ICFG, both drawn from `NewCarList$Itr`.

Phases 3 and 4 After the next two phases, the following symbolic state reaches entry:

$$c \neq \texttt{null} \wedge c.\texttt{elems} \neq \texttt{null}$$
$$\wedge \quad c.\texttt{elems.length} > 0 \wedge c.\texttt{elems}[0] \neq \texttt{null}$$
$$\wedge \quad c.\texttt{elems}[0].\texttt{year} \neq 2009 \wedge \texttt{subType}(\texttt{typeOf}(c), \texttt{NewCarList})$$

(For clarity, we write $x.\texttt{foo}$ for $\texttt{read}(\texttt{foo}, x)$, where `foo` cannot be an `update` or `mod` term.) Since this formula contains no skolem constants, it represents a path with no skipped calls. A reader may verify that this pre-condition at `entrypoint` would indeed lead the execution to goal.

Note that the order in which calls are expanded can affect performance significantly. For example, if our algorithm insisted on expanding the `next()` call at line 34 of Figure 1 first, it may have tried many possibilities before finding the method corresponding to `NewCarList`. Our implementation employs simple heuristics to determine a profitable order to expand calls, which works well in our experience. Furthermore, our implementation may heuristically expand more than one in a stage, especially calls to small methods and single-dispatch calls.

5. Enhancing Summary Reuse

As discussed in Section 2.2, a naïve approach to tabulation-based interprocedural analysis yields poor reuse of procedure summaries. Here we describe *generalization*, a technique that enables computation of generalized summaries entirely within a tabulation-based analysis [26].

For generalization, we adapt two ideas to our interprocedural *wp* framework. The first idea is to compute underapproximate symbolic summaries [10]. Whereas usual functional IPA creates tables of how *individual* input facts map to output facts, a symbolic summary applies to general *classes* of input facts.[5] Symbolic summaries can give better reuse because they are more generally applicable.

The second idea uses the frame rule [27] to analyze callees with smaller formulae. Suppose a formula ϕ_{post} propagated to a call site of m could be re-written as $\phi^I_{post} \wedge \phi^D_{post}$, in a way that no subterm of ϕ^I_{post} is written by m. ϕ^I_{post} is the method independent part, and ϕ^D_{post} is the method dependent part. Then, using the frame rule, for a call to m we have $wp(m, \phi^I_{post} \wedge \phi^D_{post}) = \phi^I_{post} \wedge wp(m, \phi^D_{post})$. The advantage of this decomposition is that a summary can be created and looked up based only on ϕ^D_{post}, and thus applied more generally.

Algorithm Our generalization algorithm rewrites a formula to a form in which the above two ideas easily apply, by generalizing terms referencing locations. Some conditional rewrite rules for generalization are in Figure 5. The l_i variables are fresh generic variables introduced to achieve a symbolic summary, and v_{ret} is

if v_{ret} occurs in ϕ :
$$\phi \rightarrow \phi[l_i/v_{ret}] \wedge l_i = v_{ret}$$
if $\texttt{read}(\texttt{f}, e)$ occurs in ϕ :
$$\phi \rightarrow \phi[l_i/\texttt{read}(\texttt{f}, e)] \wedge l_j = e \wedge l_i = \texttt{read}(\texttt{f}, l_j)$$
if $\texttt{aread}(a, e_1, e_2)$ occurs in ϕ :
$$\phi \rightarrow \phi[l_i/\texttt{aread}(a, e_1, e_2)] \wedge l_j = e_1 \wedge l_k = e_2$$
$$\wedge \, l_i = \texttt{aread}(a, l_j, l_k)$$

Figure 5. Conditional rewrite rules for generalization.

the variable assigned the return value of the call, if any.[6] After the rules are applied for all subterms possibly modified by the callee, the method-dependent conjuncts for ϕ^D_{post} are the $l_i = t$ terms where the callee may modify t.[7] Now, when the callee is analyzed—using standard tabulation—the functional IPA creates a generalized summary based only on ϕ^D_{post}.

For example, consider `id(x) { return x; }` with call site `q = id(p)` and postcondition $q > 5$. The call modifies only q, so generalization via the rewrite rules in Figure 5 yields $l_0 > 5 \wedge l_0 = q$. ϕ^D_{post} is $l_0 = q$, or $l_0 = ret$ in the callee namespace, and ϕ^I_{post} is $l_0 > 5$. Tabulation of ϕ^D_{post} through `id` builds the summary $l_0 = ret \xrightarrow{\texttt{id}()} l_0 = x$ (a fully general summary for `id`). Propagating back to the caller and applying the frame rule, we get ϕ_{pre} is $l_0 = p \wedge l_0 > 5$, which simplifies via elimination to $p > 5$, as expected.

Note that generalization entails a trade-off: fully general summaries are easier to reuse, but potentially more difficult to compute, because the generalization erases information that could be used to prune infeasible paths while analyzing the callee. Our system intentionally computes less general summaries than possible at times: it does not generalize for possibly modified locations not seen in any call site formula, and it does not generalize for post-conditions constraining the return value to be `true`, `false`, `null`, or non-`null`.

5.1 Generalization Example

Let us consider how generalization affects analysis of `test` in Figure 2; the goal again is to find a precondition leading to an exception at line 38 of Figure 1. For brevity, we elide various irrelevant conjuncts throughout this discussion. As shown in Section 4, the analysis discovers a pre-condition for `entrypoint` with the conjunct $c.\texttt{elems}[0].\texttt{year} \neq 2009$, which before line 8 of Figure 2 becomes

$$l.\texttt{elems}[0].\texttt{year} \neq 2009 \qquad (1)$$

For the `setYear` call at line 7, since the callee may modify `year`, generalization yields

$$l_0 \neq 2009 \wedge l_0 = l_1.\texttt{year} \wedge l_1 = l.\texttt{elems}[0] \qquad (2)$$

We propagate the only possibly-modified conjunct, $l_0 = l_1.\texttt{year}$, to the callee. Analysis of `setYear` yields the summary

$$l_0 = l_1.\texttt{year} \xrightarrow{\texttt{setYear}()} l_0 = \texttt{read}(\texttt{update}(\texttt{year}, \texttt{this}, \texttt{y}), l_1) \qquad (3)$$

which is fully general for `setYear`. Applying this summary edge via the frame rule at line 7 (substituting actuals for formals) yields

$$l_0 \neq 2009 \wedge l_0 = \texttt{read}(\texttt{update}(\texttt{year}, c_2, 2009), l_1) \wedge l_1 = l.\texttt{elems}[0] \qquad (4)$$

which, after elimination of generic variables, further simplifies to

$$c_2 \neq l.\texttt{elems}[0] \wedge l.\texttt{elems}[0].\texttt{year} \neq 2009 \qquad (5)$$

---

[5] Note the distinction between tabulation-based summaries in a domain of symbolic formulae and symbolic summaries in this general sense as described in [10].

[6] We elide similar rules and handling of `update` and exception variables for brevity.

[7] Our implementation currently uses a simple type-based analysis to reason about possibly modified locations. Sharper analysis is possible, but it would be more expensive to compute. We will examine the tradeoff in future work.

(if $c_2 = l.$elems$[0]$, the formula becomes `false`).

Now the analysis reaches the `set` call at line 6. Since `set` modifies the contents of an array (`this.elems`), generalization of (5) yields

$$l_0 = l_1[l_2] \wedge l_1 = l.\texttt{elems} \wedge l_2 = 0 \wedge c_2 \neq l_0 \wedge l_0.\texttt{year} \neq 2009 \tag{6}$$

Analyzing `set` with the possibly modified conjunct $l_0 = l_1[l_2]$ yields the summary edge

$$l_0 = l_1[l_2] \xrightarrow{\texttt{set()}} \tag{7}$$
$$l_0 = \texttt{aread}(\texttt{aupdate}(a, this.\texttt{elems}, i, c), l_1, l_2)$$

When applied at line 6 with $i = 1$ and $l_2 = 0$ (different indices), the `aread` term simplifies to $l.$elems$[0]$. Hence, the formula before line 6 is (5), unmodified by the call. Analyzing line 5 shows $c_2 \neq l.$elems$[0]$, yielding formula (1) again before line 5.

We shall now see reuse of the above summary edges. Generalization of (1) for the `setYear` call at line 4 again yields (2), so clearly summary edge (3) can be reused for the call. Applying the edge yields

$$\texttt{read}(\texttt{update}(year, c_1, 2008), l.\texttt{elems}[0]) \neq 2009 \tag{8}$$

before line 4. Generalizing (8) for the `set` call at line 3 yields

$$l_0 = l_1[l_2] \wedge l_1 = l.\texttt{elems} \wedge l_2 = 0 \tag{9}$$
$$\wedge \texttt{read}(\texttt{update}(year, c_1, 2008), l_0) \neq 2009$$

Since the first conjunct is identical to that of (6), we can reuse summary edge (7) for this call. In this case, we have $i = l_1 = 1$ at the caller, so the `aread` term from (7) simplifies to `c1`. Hence, we have

$$\texttt{read}(\texttt{update}(year, c_1, 2008), c_1) \neq 2009$$

which simplifies to `true`. This means `test` always causes an exception at line 38 of Figure 1, as expected.

6. The Rest of the Story

As with any non-trivial system, many design decisions strongly impact the real-world performance of SNUGGLEBUG. Here, we briefly discuss other important aspects of the system.

Disjunctive formula propagation The cornerstone of our design is to propagate minterms—the disjuncts of a formula in disjunctive normal form (DNF)—independently during wp computation. This separate propagation is possible due to the distributivity property of wp, $wp(s, \phi_a \vee \phi_b) = wp(s, \phi_a) \vee wp(s, \phi_b)$. This design gives rise to the set of symbolic states $D(s)$ at each program point in Figure 3; in the implementation, each formula in the set is a minterm.

Propagating minterms independently has several advantages. First, it exposes redundant states that need not be explored separately. For example, if $D(s)$ contains distinct disjuncts A, B, and C, we need only propagate A, B, and C independently, not disjunctive combinations like $A \vee B$. For similar reasons, propagating minterms improves reuse of summary edges during interprocedural propagation. Additionally, independent minterms tend to stay small—even as there can be more of them— which makes simplification less costly. Instead of repeatedly performing expensive conversions to DNF, the implementation of each wp transformer outputs a set of minterms.

On-the-fly simplification Many formulae develop internal contradictions and become unsatisfiable during wp computation; such formulae must be dropped from propagation early for good performance. One way to detect contradictions is to use a full SMT solver at each propagation step, but we found this to be too expensive. SNUGGLEBUG, therefore, incorporates a custom, lightweight

$$(x \leq y) \wedge (x \neq y) \rightarrow x < y$$
$$(x < y) \wedge (x \neq y) \rightarrow x < y$$
$$0 \leq \texttt{read}(length, x) \rightarrow \texttt{true}$$
$$(\texttt{typeOf}(a) = T) \wedge (a = \texttt{null}) \rightarrow \texttt{false}$$
$$\texttt{subType}(a, c1) \wedge \texttt{subType}(a, c2) \wedge \texttt{subType}(c1, c2) \rightarrow \texttt{subType}(a, c1)$$
$$(\texttt{typeOf}(x)\ subtype\ T) \wedge (\texttt{isFinal}(T)) \rightarrow \texttt{typeOf}(x) = T$$
$$\texttt{read}(f, fresh(T)) \rightarrow \delta(T)$$

Figure 6. Sample rewrite rules in our on-the-fly simplifier. `isFinal` is a predicate which identifies `final` Java classes. $\delta(T)$ denotes the default value for a type in a newly allocated field.

simplifier, serving the `simplify()` function in Figure 3. Our simplifier relies on a difference constraints solver, a term rewriting engine, and standard elimination techniques to compute solved forms of constraint systems.

Figure 6 shows some of SNUGGLEBUG's rewrite rules. Additional rules (not shown) simplify terms involving method dispatch, arithmetic, theory of arrays, reflection, etc. Additionally, we canonicalize formulae via hash consing and fold constants during term construction.

Loops, Recursion, and Search Heuristic The algorithm presented in Section 3.2 does not enforce fixed upper bounds on the number of times loops and recursion are analyzed; without such bounds, the algorithm may not terminate. SNUGGLEBUG instead works with a generous but fixed budget on the number wp steps for each goal, aborting thereafter.

For typical SNUGGLEBUG applications, we just need one satisfactory precondition to reach the program entry. The selection of which item in the worklist to process next—*i.e.* the search strategy—has a big impact on whether the budget is used well. In fact, it can be shown that in the presence of common forms of loops, random selection of items from the worklist can lead to pathological behavior, squandering away the budget. As observed in work on symbolic execution [31, 9]—and well-understood for graph search algorithms in general—an *informed* search heuristic is key to performance for such an approach.

Our current search heuristic prioritizes paths with less looping or call depth, searching for a feasible path in a quasi breadth-first manner. A secondary heuristic prioritizes program points that fewer facts have reached, aiming to spread the search effort more equitably across the program. A rigorous study of search heuristics in SNUGGLEBUG is a topic for future work.

Constraining Search with Overapproximation As noted in other work [19], abstract interpretation can aid symbolic analysis by cheaply computing overapproximate information. We use an *ad hoc* pre-pass of abstract interpretation to determine certain invariants such as constants, known array lengths, and non-nullness. The symbolic analysis uses these invariants to simplify formulae and prune infeasible paths.

More importantly, overapproximate analysis will be a key tool necessary to synthesize loop invariants, enabling better handling of loops; this is a topic of future work for us.

Other details We implemented our algorithm using the T.J. Watson Libraries for Analysis (WALA) [32]. The interprocedural analysis is built on WALA's tuned tabulation solver [26], aiding scalability. The implementation handles all features of (sequential) JVM bytecode semantics, including intra- and inter-procedural exceptional control flow, string constants, and reflection via class objects manipulated with `ldc`. We additionally handle many native

Benchmark	Version	Source kLOC
`ant`	1.7.0	88
`antlr`	2.7.2	38
`batik`	1.6	157
`tomcat`	6.0.16	163
`eclipse.ui`	3.3.1	305

Table 3. Information about our benchmarks, popular open-source Java programs. `eclipse.ui` consists of the plugins from Eclipse in the `org.eclipse.ui.*` namespace.

Configuration	Validated	Not Validated	Avg. Time Per Goal (s)
Production	29	0	93
NoGeneralization	27	2	193
NoSimplification	11	18	1122
NoFeedback-CHA	12	17	1057
NoFeedback-Andersen	8	21	1331

Table 4. Comparison of results across five configurations.

methods from the standard libraries with models, including many features of reflection.

Our analysis does not reason about concurrency. The implementation does not model exactly the semantics of some bitwise operators, floating point arithmetic, and integer overflow issues. For any native method m in a client program, we assume that (1) m cannot throw an exception, (2) m can return an arbitrary value, and (3) m does not modify the heap.

When SNUGGLEBUG pushes a symbolic state to an entrypoint, it uses the SMT solver CVC3 [6] as a final check of satisfiability. In practice, most of the satisfiability queries are decidable. However, our logic can encode nonlinear constraints over integers, which are undecidable. The solver may return "unknown" in such cases; the tool interprets this result as "unsat" and continues searching.

7. Evaluation

We evaluated our algorithm by using it to validate null dereference warnings output by FindBugs [21]. As noted in Section 1, bug validation is just one possible use of SNUGGLEBUG; we chose this client for the evaluation to obtain an unbiased source of goals for our analysis. Also, note that SNUGGLEBUG-style sound analysis may not present the best tradeoffs for bug validation; a real-world tool may choose to trade off some soundness (*e.g.*, by assuming some methods do not throw exceptions) for better performance.

We consider a report of a possible null-pointer exception (NPE) at program point p *validated* if our analysis discovers a precondition ϕ for the closest *public* method m such that if m is invoked with parameters satisfying ϕ, p *must* throw an NPE. The emphasis on public methods is deliberate: a potential problem in a private method may prove infeasible if all callers of the private method establish an appropriate invariant. Hence we try to establish interprocedural feasibility from a public method, which makes for a more useful client, but also for much more challenging analysis.

Note, that our validation procedure does *not* verify that a state satisfying the precondition ϕ can actually be constructed via the public APIs of the corresponding classes. Other work has addressed test generation respecting such invariants [11], but such issues fall outside the scope of this paper.

We ran FindBugs v.1.3.4 on a number of open-source Java programs, and selected for this experiment those in which Find-Bugs reported potential null pointer bugs, shown in Table 3. Together, these benchmarks comprise more than 750,000 lines of non-comment non-whitespace code, with millions of lines of dependent libraries.

To measure the effectiveness of our techniques, we ran the following five analysis configurations:

Production All described techniques enabled.

NoGeneralization Generalization (Sec. 5) disabled.

NoSimplification On-the-fly simplification (Sec. 6) disabled.

NoFeedback-CHA Analysis over a pre-computed call graph based on class hierarchy analysis instead of directed call graph construction (Sec. 4)

NoFeedback-Andersen Analysis over a pre-computed call graph based on Andersen's analysis [2] instead of directed call graph construction (Sec. 4)

All experiments ran on a dual-processor IBM ThinkCentre running Windows XP, with two 3GHz Pentium 4 processors and 2GB of RAM. Our analysis infrastructure (described in Section 6) ran on the Sun JDK 1.6 with 1GB of max heap space.

7.1 Results

Does the analysis work for large programs? From the 750,000 lines of source code considered, FindBugs reported findings for 56 possible NPE bugs in the considered categories. We examined each finding by hand and with SNUGGLEBUG. SNUGGLEBUG successfully validated 29 of the 56 findings (52%).

Of the remaining 27 findings, we concluded based on manual inspection that 18 are infeasible, *i.e.*, there is no feasible path from a public entrypoint that results in the NPE. Infeasibility in all of these cases was due to some invariant enforced interprocedurally.

The remaining 9 findings represent cases that we believe to be feasible, but SNUGGLEBUG could not find a valid precondition within the allotted time (30 minutes). Overall, SNUGGLEBUG succeeded in finding a precondition for 29 of 38 feasible cases (76%). SNUGGLEBUG failed to validate a few cases due to non-linear arithmetic beyond the reach of the SMT solver, inability to reason about type conversions between integers and unsigned ints and floats, and an incomplete model of native methods related to the Java security model. The remaining cases presented a bigger search space than SNUGGLEBUG could handle in the time limit, often requiring analysis of complex XML parsing libraries.

The remainder of this section evaluates particular techniques presented in this paper. We restrict our attention to the 29 validated findings for the remainder of this section, since the other cases time out on all configurations.

Table 4 compares results across the five configurations. The second column shows the number of bugs validated by each configuration; the remaining tasks timed out with the 30 minute limit. The last column reports the arithmetic mean of running time for each task. This time represents the end-to-end wall-clock time, including call graph construction, refinement, and calls to the SMT solver. When tasks time out, we assign a time of 30 minutes; so when timeouts occur, the reported time is a lower bound.

Figure 7 presents more details on the distribution of times for the 29 tasks considered. The figure shows for each configuration, the percentage of tasks completed as a function of time. The results show that over half the tasks are "easy", in that Production validates the goal in one minute or less. The Production configuration validated each of the 29 goals in 13 minutes or less. NoSimplification, NoFeedback-CHA and NoFeedback-Andersen seem effective only for some "easy" tasks; these configurations discover no preconditions after three minutes of analysis.

How effective is directed call graph construction? The last two rows of Table 4 consider results with pre-computed call graphs instead of directed call graph construction. The NoFeedback-CHA
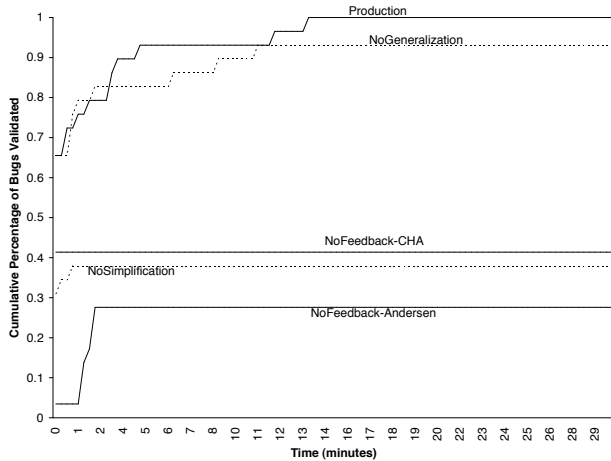
Figure 7. Comparison of running times across configurations.

Figure 8. Effective call graph sizes.

configuration builds a call graph with class hierarchy analysis; this call graph is conservative but imprecise.

The NoFeedback-Andersen configuration builds a call graph on-the-fly with context-insensitive Andersen's pointer analysis [2], as implemented in WALA [32]. The WALA implementation handles many difficult language features, including reflection patterns and models of many native methods. We initialize points-to sets for parameters to entrypoints with objects of all possible parameter subtypes non-deterministically, which may still be incomplete due to missing subtypes for parameter fields. With WALA's relatively conservative treatment of reflection, call graph construction for large programs on the Java 1.6 libraries would exceed the 30 minute timeout; we limited the call graph construction to 25K nodes, which ran to completion for all cases in a couple of minutes.

Table 4 shows that both configurations with pre-computed call graphs are ineffective, failing to validate most of the tasks within the time limit and degrading performance by at least a factor of ten. As just discussed, our best-effort Andersen call graph is potentially unsound due to difficulties with pointer analysis and library entrypoints; this accounts for the 4 cases where NoFeedback-Andersen failed to validate a bug found by NoFeedback-CHA.

Figure 8 presents data illustrating the size of the program subset explored while validating the call graph. The figure shows that with directed call graph construction, roughly 50% of the tasks explored a call graph of 10 nodes or less. The largest expanded call graph contained 93 nodes.

For the pre-computed call graphs, we define an *Effective Call Graph* as follows. For each task, let d be the maximum depth of the expanded call graph built by the Production configuration. Given a pre-computed call graph G and an entrypoint e, we define the Effective Call Graph to consist of those nodes in G which are reachable from e via breadth-first-search (BFS) up to depth d. Assuming an oracle provided the necessary depth d, the Effective Call Graph is the smallest known to be sufficient for the task.

Figure 8 shows that the effective call graph using Andersen's analysis is typically a factor of ten larger than the one built by directed construction, and the class-hierarchy call graph is at least a factor of ten larger still. We conclude that directed call graph construction is crucial.

How effective is our summary-based reuse? We measured summary edge reuse as follows: each time a fact propagates interprocedurally to a callee, we record whether or not a summary edge already exists; we call the percentage of times a summary edge already exists the *reuse factor*. Across all runs, the Production con-
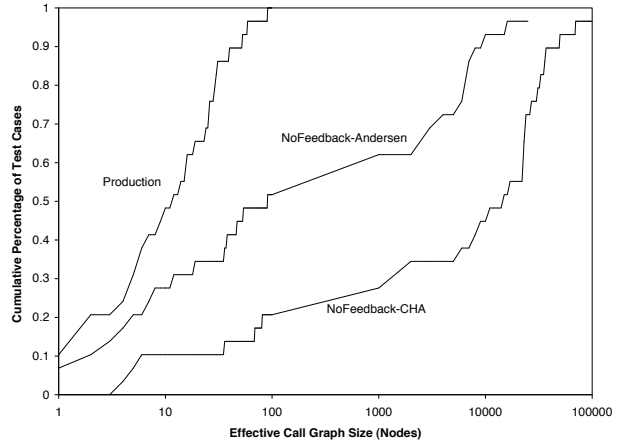
```
1  public void setPrefix(String prefix) throws DOMException {
2    if (this.isReadonly()) {
3      throw createDOMException(DOMException.NO_MOD_ALLOWED_ERR,
4        "readonly.node",
5        ...);
6    }
7    String uri = this.getNamespaceURI();
8    if (uri == null) {
9      throw createDOMException(DOMException.NAMESPACE_ERR,
10       "namespace",
11       ...);
12   }
13   String name = this.getLocalName();
14   if (prefix == null) {
15     this.setNodeName(name);
16   }
17   if (!prefix.equals("") &&
18       !DOMUtilities.isValidName(prefix)) {
```

Figure 9. Excerpt from batik-0.6, AbstractNode.java

figuration saw a reuse factor of 87%, while the NoGeneralization configuration saw a reuse factor of 5.3%. Table 4 shows that generalization improves performance by more than a factor of two (93s vs. 193s, on average). We conclude that tabulation-based modular analysis *with generalization* is effective for interprocedural propagation of weakest preconditions.

Note that although our reuse factor is already 87%, it is possible that increasing reuse further by a small amount could have a large impact on bottom-line performance, *e.g.*, if repeated analysis of large methods were avoided. Also, there could be further large performance benefits from reusing work across phases of directed call graph construction, as discussed in Section 4.

What is the impact of on-the-fly simplification? Table 4 reports that without on-the-fly simplification, performance degrades by at least a factor of 10, and 18 of the 29 tasks fail to finish within 30 minutes. As with pre-computed call graphs, Figure 7 suggests that NoSimplification is effective only for "easy" tasks, making no further progress after the first two minutes of analysis.

7.2 Examples from Experiments

In this section, we offer selected short code excerpts from the experimental study, in order to illustrate how the concepts discussed in this paper arise in real-world examples.

Example from Apache Batik Figure 9 shows an excerpt from batik-0.6, an instance method from class AbstractNode. The

```
1  public String[] findManagedBeans(String group) {
2    ArrayList results = new ArrayList();
3    Iterator items = descriptors.values().iterator();
4    while (items.hasNext()) {
5      ManagedBean item = (ManagedBean) items.next();
6      if ((group == null) && (item.getGroup() == null)) {
7        results.add(item.getName());
8      } else if (group.equals(item.getGroup())) {
9        results.add(item.getName());
10     }
11   }
12   String values[] = new String[results.size()];
13   return ((String[]) results.toArray(values));
14 }
```

Figure 10. Excerpt from tomcat-6.0.16, Registry.java

goal is to reach line 17 with `prefix` equal to `null`. It is easy to see that `prefix` must be `null` at entry to reach this goal. However, a sound analysis must find a precondition that also satisfies the following conditions:

1. The virtual call to `isReadonly()` (line 2) must return false to proceed beyond the first conditional.

2. The virtual call to `getNamespaceURI()` (line 7) must return a non-null value to continue past the second conditional.

3. The virtual calls to `getLocalName()` (line 13) and `setNodeName()` (line 15) must then return without throwing an exception to continue to the goal.

4. The type of `this` must be such that the concrete methods to which the above calls dispatch together exhibit the required behavior.

The `batik` source code includes 85 concrete subtypes of `AbstractNode`, with dozens of implementations of the virtual methods just described. Only a few subtypes can be instantiated as `this` and satisfy these criteria. It is difficult for a human to examine all 85 types and reason about whether any such type satisfies these criteria. [8]

Our analysis with pre-built call graphs failed to find a satisfiable precondition, since the conservatively computed call graphs expose a search space that is too big. However, the directed call graph construction succeeded in narrowing the search for the appropriate subtype of `AbstractNode`, relying on type and dispatch constraints handled by the theorem prover to rule out types associated with (dead-end) method implementations explored in early iterations.

Example from Apache Tomcat   Figure 10 shows an example from Apache Tomcat, similar in spirit to the motivating example of Figure 1. The goal is to find a precondition such that execution reaches line 8 with `group == null`.

Clearly `group` must be `null` at method entry to reach the goal. However, other conditions also must be satisfied:

1. The `descriptors` field (of declared type `HashMap`) of `this` must be non-null, and `descriptors.values().iterator()` must return a non-null, non-empty `Iterator`.

2. The `items.next()` call must return a non-null `item`, and `item.getGroup()` cannot return `null`.

Note that it is impossible for a human to determine whether this condition is feasible from inspecting just this code snippet; one must also check that the conditions on callees (such as `getGroup`) are satisfiable.

---

[8] In fact, when first looking at this example, we hypothesized that the condition was infeasible.

Analysis of this example requires the same type of reasoning as previously discussed for Figure 1. Through several iterations, the analysis determines appropriate `Set` and `Iterator` implementations that are consistent with these conditions, and verifies that that iterator can return a `ManagedBean` such that `item.getGroup() != null`.

8. Related Work

Backward symbolic analysis   ESC/Java [15] pioneered practical weakest-precondition analysis for Java. SNUGGLEBUG follows ESC in performing abstraction-free, underapproximate backwards symbolic reasoning, checking satisfiability with a theorem prover. ESC generated verification conditions of worst-case quadratic size, pushing exponential search factors into the theorem prover. In contrast, SNUGGLEBUG manages the exponential search space directly, allowing for path pruning via inexpensive on-the-fly simplification (Sec. 6).

ESC employed only intraprocedural analysis, relying on user annotations and specifications to reason across procedure calls. To reduce the annotation burden, the ESC/Java team developed Houdini [14], a tool to infer specifications for unannotated programs. Houdini generated a large set of candidate specifications for each procedure and checked them using ESC/Java. In contrast, SNUGGLEBUG follows a more direct approach to generating procedure summaries driven by functional-style IPA [29].

Boogie [5] is a verifier for Spec# in the tradition of ESC. The Boogie program representation includes *modifies* specifications for procedures, which SNUGGLEBUG could exploit to improve separation. Boogie also employs abstract interpretation to synthesize loop invariants, also potentially beneficial to SNUGGLEBUG.

Like SNUGGLEBUG, the PSE tool [24] performed backwards interprocedural symbolic analysis using functional-style IPA, targeting bug validation. Unlike SNUGGLEBUG, PSE did not provide a sound underapproximate analysis, since the tool did not represent the entire path condition and sometimes fell back to abstract representations of the heap.

Preconditions generated by symbolic analysis to a library entry-point may still be infeasible, ruled out by other program invariants. Addressing this problem, the DSD-Crasher [11] tool combines ESC-Java with a tool to generate concrete tests as counterexamples. Moreover, it uses a dynamic invariant detector to constrain inputs to obey likely invariants, ruling out some spurious bug reports. Handling this invariant issue in SNUGGLEBUG remains for future work.

Program Testing via Symbolic Execution   Program testing generally targets high coverage, and not specific goal states. Nevertheless, much work that enhances program testing using symbolic execution has technical connections to SNUGGLEBUG.

PREfix [8] was among the first systems to show fully automatic interprocedural symbolic execution on real-world programs. PREfix relied on a pre-built call graph and processed procedures bottom-up, building underapproximate procedure summaries. These summaries may not represent all possible input environments to a function, so PREfix falls back to conservative estimates of behavior as needed. In contrast, SNUGGLEBUG computes partial summaries as needed on demand.

Saturn [33] performs an interprocedural bit-precise symbolic execution for C programs, and it has been demonstrated to find bugs in large systems programs. Saturn employs property-specific function summaries for modular interprocedural analysis, but the summaries do not generally support full interprocedural path sensitive analysis. In contrast, SNUGGLEBUG builds fully automatic path-sensitive symbolic summaries on demand, exploiting generalization to enhance reuse.

Calysto [3] introduced "structural" abstraction, a staged symbolic execution that initially skips over many calls and inlines them later if they appear on feasible paths. This work represents the closest precursor to our directed call graph construction. Calysto relied on a pre-computed call graph for handling indirect calls; in contrast, our directed call graph construction exploits constraints discovered during symbolic analysis and requires no whole-program pointer analysis. Calysto did not perform summary-based interprocedural analysis, but instead inlined callee representations.

Systems such as DART [17] and CUTE [28] use symbolic analysis in concert with concrete execution to improve coverage of random testing. To address scalability problems with the DART-like approach, the SMART system [16] incorporated function summaries. Follow-on work [1] described a demand-driven computation of summary edges, similar in spirit to the tabulation algorithm [26] we use. This work [1] also described the use of uninterpreted functions to skip processing of callees.

KLEE [9] and Java PathFinder [31] perform interprocedural symbolic execution, each with novel techniques for aggressive on-the-fly simplification and path pruning, optimized representations of the symbolic search space, and informed search heuristics. SNUGGLEBUG builds on this approach, applying analogous techniques to backward symbolic analysis of Java.

Jackson and Vaziri [22] translate a program into constraints in the Alloy specification language, finitizing the problem via bounded unrolling of loops, method inlining, and bounding the heap. Miniatur [13] extended this work, using program slicing to reduce the search space. Demand-driven backwards analysis obviates the need for such slicing. Work by Taghdiri [30] extended this approach to avoid inlining callees, instead inferring partial method specifications via abstraction refinement.

Abstraction-based approaches Counter-example guided abstraction refinement (CEGAR) systems [4, 20] perform abstract interpretation over a predicate abstraction and refine predicates based on feedback from a precise symbolic execution. These systems could be used to concretize paths to errors, such as the experiment we considered. Directed call graph construction can be viewed as a form of CEGAR.

Synergy [19] presented an algorithm to combine CEGAR analysis with DART-like underapproximate search. Dash [7] extended Synergy to the interprocedural setting and introduced an algorithm to refine pointer abstractions without whole-program pointer analysis. In general, CEGAR approaches and SNUGGLEBUG are complementary: SNUGGLEBUG may benefit from a feedback loop with abstraction refinement, and CEGAR scenarios could benefit from directed call graph construction.

9. Conclusion

We have presented SNUGGLEBUG, a new approach to demand-driven interprocedural symbolic analysis. We presented several novel techniques to improve scalability, including directed call graph construction, generalization to improve reuse, and lightweight domain-specific on-the-fly simplification and path pruning. Results for bug validation tasks on large Java libraries indicate that the techniques work in concert to improve performance significantly, bringing practical tools based on this technology within reach.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, 2008.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[3] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *ICSE*, 2008.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[5] M. Barnett, B. E. Chang, R. Deline, B. Jacobs, and K. R. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.

[6] C. Barrett and C. Tinelli. CVC3. In *CAV*, 2007.

[7] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA*, 2008.

[8] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[10] P. Cousot and R. Cousot. Modular static program analysis. In *CC*, 2002.

[11] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM TOSEM*, 17(2):1–37, 2008.

[12] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[13] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In *FSE*, 2007.

[14] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, 2001.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[16] P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[18] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[19] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *FSE*, 2006.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, 2004.

[22] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA*, 2000.

[23] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[24] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. *FSE*, 2004.

[25] J. McCarthy. A basis for a mathematical theory of computation. Technical report, MIT, Cambridge, MA, USA, 1962.

[26] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[27] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE*, 2005.

[29] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[30] M. Taghdiri. Inferring specifications to detect errors in code. *Automated Software Engineering, International Conference on*, 0:144–153, 2004.

[31] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA*, 2004.

[32] T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net.

[33] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM TOPLAS*, 29(3):16, 2007.