Lightweight Extraction of Syntactic Specifications

Mana Taghdiri Robert Seater Daniel Jackson
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{taghdiri, rseater, dnj}@mit.edu

ABSTRACT

A method for extracting syntactic specifications from heap-manipulating code is described. The state of the heap is represented as an environment mapping each variable or field to a relational expression. A procedure is executed symbolically, obtaining an environment for the post-state that gives the value of each variable and field in terms of the values of variables and fields of the pre-state. Approximation is introduced by forming relational unions at merge points in the control flow graph, and by widening union-of-join expressions to transitive closures. The resulting analysis is linear in the length of the code and the number of fields, but capable of producing non-trivial specifications of surprising accuracy.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; F.3.1 [**Logics**]: Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms

Verification

Keywords

Modular Abstraction, Symbolic Summary, Symbolic Execution, Syntactic Specification

1. INTRODUCTION

Analyzing a large program is not usually feasible unless the program is decomposed into smaller parts that can be analyzed separately, obtaining results that can be combined to determine the behavior of the program as a whole. Such results are called 'summaries' in the static analysis community, and are usually associated with procedures; this makes sense both because the procedure is a unit of reuse, and because its encapsulation properties allow for more succinct summaries.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.

A procedure summary is typically a semantic object produced and consumed within the analysis tool, and never exposed to the user. In contrast, a type signature – the form of summary used by a type analysis (e.g. [4]) – is a syntactic object that is readable by tool and user alike, and can play exactly the same role as a specification, associated by the programmer with a procedure to enable modular reasoning.

This paper describes a method for extracting such syntactic summaries, or specifications, from code in languages such as Java or C# whose state can be represented as a graph of objects linked by fields. Our original motivation was to support a code analysis tool based on constraint solving [15]. This tool infers specifications (of a less readable form) incrementally from code, refining them in response to spurious counterexamples. Although it can start from arbitrarily weak specifications (including the empty specification), its performance is dramatically improved if the initial specification captures at least some basic frame conditions about which objects and fields might be mutated. In devising a method to obtain these simple specifications, we realized that a very simple and scalable analysis can in fact produce far more accurate specifications which could be used in a variety of settings.

The key idea behind the method is very simple. The procedure is evaluated symbolically, with each field and variable initially holding a value represented by a fresh constant. Each statement updates an environment mapping fields and variables to values represented as relational expressions. After an if-statement, the branches are merged by forming a union expression. A loop is handled by forming a transitive closure. The analysis maintains both upper and lower bounds on the values of variables and fields, so that in some cases an exact result can be obtained.

This analysis can be viewed as a simple abstract interpretation in which the abstract domain contains relational expressions. The union of two expressions approximates each expression from above, justifying the merge. Two kinds of widenings are performed – turning unions into closures, and replacing an expression by the universal relation – to guarantee that the size of any expression (and the number of loop iterations required to reach a fixpoint) is bounded by a constant. Consequently, the cost of the analysis of a single statement is proportional to the number of fields, and the analysis time is linear in both the length of the code and the number of fields. The analysis space is linear in the number of fields and the maximum number of local variables in scope at any point, and is constant in the length of the code.

To evaluate the method, we performed two experiments

using a prototype tool implementation. In one, we applied it to a small component for which complete specifications were already available: Sun's standard implementation of linked lists in the Java Collection Framework. The procedures of this component are small and mostly self-contained; their complexity arises from the manipulations they perform of the doubly-linked data structure. Despite this complexity, the tool generated surprisingly accurate specifications. In 8 of the 10 procedures, the generated specifications were sufficient to check the known JML specifications; in the remaining 2 procedures, the generated specifications included all frame conditions, and provided some parts of the JML specifications.

In the second experiment, we applied the tool to an open-source Java implementation of a graph API [9]. This component was more typical of object-oriented code; its procedures made more external calls – in one case, a top-level procedure invoked 81 other procedures directly or indirectly – but it used simpler data structures, primarily sets and hash tables accessed as abstract data types. We wrote specifications for these datatypes in the style of the specifications our tool infers; this allowed the analysis to be performed over specification fields rather than the actual representation fields of the implementation. Lacking a full specification of the graph API, we used some partial invariants to evaluate the accuracy of our generated specifications, and we found that only in a few cases was there a significant loss of information. These results suggest that this analysis represents a useful balance between tractability and accuracy.

In summary, the contributions of this paper include:

- A new approach to extracting lightweight specifications from heap-manipulating code, consisting of a simple relational specification language and a cheap static analysis;
- A justification of the soundness of the analysis formulated as an abstract interpretation;
- A demonstration that the analysis is capable of extracting reasonable specifications from non-trivial procedures;
- A demonstration that the analysis can leverage user-provided specifications of libraries, and generate specifications over abstract rather than concrete fields.

The remainder of the paper is organized as follows: Section 2 gives an overview of the method, describes the underlying logic, and provides some small examples. Section 3 gives the technical details of the analysis. Section 4 presents our experimental results. Section 5 compares our method with the related work, and Section 6 concludes.

2. OVERVIEW

Given a procedure in an object-oriented program, our analysis summarizes the behavior of that procedure by computing a symbolic relationship between the procedure's pre-states and post-states. Our summaries are declarative formulas written in a relational logic with transitive closure. A summary gives both an over- and an under-approximation of the procedure's behavior. It bounds the final values of accessed fields, the return value, and allocated objects of the analyzed procedure by symbolic relational expressions.

```
Stmt ::= Var = PExpr
       | PExpr.Field = PExpr
       | Var = new Class(PExpr*)
       | Proc(PExpr*)
       | Var = Proc(PExpr*)
       | return PExpr
       | if (Cond) Stmt [else Stmt]
       | while (Cond) Stmt
       | Stmt; Stmt
PExpr ::= Const | Var[.Field]*
Const ::= null | true | false
Cond  ::= Var[.Field]*== PExpr
       | Var[.Field]*!= PExpr
       | Cond && Cond | Cond || Cond
```

Figure 1: Program statements

We abstract the pre-state of a procedure by representing each type, variable, and field as a relation with some symbolic value. The effects of each program statement on the abstract state are computed using a flow-sensitive, context-sensitive analysis based on the abstract interpretation framework [3]. For each relation, two expressions are computed: a lower bound, representing the tuples that the relation contains in *all* executions of the procedure (encoding the procedure's *must* side-effects), and an upper bound, representing the tuples that the relation *possibly* contains in some execution (encoding the procedure's *may* side-effects).

The constraints place no restrictions on aliasing between symbolic names, and thus the results account for all possible aliasing in the pre-state.

In order to generate more precise summaries, our analysis exploits the condition of a loop in abstraction of the loop body: upon entering a loop, a relational encoding of the loop condition is intersected with the expressions computed for the variables used in that condition. Thus, those tuples that violate the loop condition will be removed. The loop is then abstracted by computing a fixpoint. Then, a relational encoding of the termination condition (negation of the loop condition) is intersected with the final expressions of the condition's variables.

Our analysis is further optimized by applying a pattern matching that recognizes the loops iterating over all the elements of a linked data structure, a common pattern in heap-manipulating programs. It generates a more accurate specification for any loop that follows this pattern.

A collection of simplification rules and widenings reduces the size and complexity of the generated expressions. For example, transitive closure can often be used to concisely summarize a relation updated in a loop without losing soundness. We limit the size of relational expressions by placing an upper bound on the number of operators it can contain. Any overconstraint exceeding that bound is widened to the universal relation of the appropriate type. Any underconstraint exceeding that bound is approximated by the empty set.

2.1 Programs

The focus of our analysis is to summarize the effects of an object-oriented program on the structure of the heap. We currently support a subset of Java statements that does not include exceptions, concurrency, and arithmetic expressions. Figure 1 gives a grammar for supported program statements.

```
Summary ::= Constr | Summary ∧ Summary
Constr  ::= PostType Op AllocExpr
     | Return Op ValueExpr
     | PostField Op FieldExpr
Op ::= ⊆ | ⊇ | =
AllocExpr ::= PreType | SymObj | SymObjSet
     | AllocExpr + AllocExpr                union
     | AllocExpr & AllocExpr                intersection
ValueExpr ::= {<Const>} | Param | SymObj
     | SymObjSet | {} | PostType
     | ValueExpr.FieldExpr                  composition
     | FieldExpr.ValueExpr                  composition
     | ValueExpr & ValueExpr                intersection
     | ValueExpr + ValueExpr                union
     | ValueExpr - ValueExpr                difference
FieldExpr ::= PreField
     | ValueExpr → ValueExpr                product
     | FieldExpr.FieldExpr                  composition
     | FieldExpr & FieldExpr                intersection
     | FieldExpr + FieldExpr                union
     | FieldExpr - FieldExpr                difference
     | FieldExpr ++ FieldExpr               override
     | *FieldExpr                           closure
PreField  ::= BinaryRelation
PostField ::= BinaryRelation
PreType   ::= UnaryRelation
PostType  ::= UnaryRelation
Return    ::= UnaryRelation
Param     ::= UnaryRelation
SymObj    ::= UnaryRelation
SymObjSet ::= UnaryRelation
```

Figure 2: Relational logic for summaries

2.2 Summaries

Our analysis encodes a procedure's behavior in a relational logic with transitive closure. Each class C declared in the analyzed program is encoded by a unary relation (a relation of arity 1) that represents the set of all allocated objects of type C. A program variable is encoded by a singleton unary relation that represents the value of that variable, and a field of type T declared in a class C is encoded by a functional binary relation of type $C \rightarrow T$ that maps each object of type C to an object of type T.

Figure 2 gives a grammar for the underlying relational logic, a subset of the Alloy modeling language [6]. For an analyzed procedure p, the summary is a conjunction of constraints for the final values of p's accessed fields, return value (if any), and allocated objects. PreField and PostField represent the relations encoding the value of a field before and after the execution of p, respectively. Similarly, PreType and PostType represent the relations encoding the initial and final sets of allocated objects of a type. The relation Return represents the return value of p.

A constraint can bound the final value of a relation from either above or below. For a relation r and a relational expression e, the subset constraint $r \subseteq e$ expresses that each tuple of r is contained in e. Similarly, the superset constraint $r \supseteq e$ expresses that r contains all the tuples in e. We use the equality constraint $r = e$ when both $r \subseteq e$ and $r \supseteq e$ hold.

A summary bounds the set of the allocated objects of each type by an allocation expression. An allocation expression for a type is a unary relational expression that can contain the initial objects of that type (PreType) and some newly allocated symbolic objects (SymObj). A summary enumerates the first m objects allocated by the analyzed procedure or its callees. Any further allocations are approximated by a symbolic set of objects (SymObjSet) whose cardinality is unspecified.

The values of program variables and fields are bounded by value expressions and field expressions, respectively. A value expression gives a unary relation that can represent a program constant, a formal parameter of the analyzed procedure, or an allocated object. It can also represent the empty set, or the set of all objects of some type. Furthermore, a value expression can be the relational composition (relational join) of a value expression and a field expression. The composition of relations r (arity n) and s (arity m) is defined as follows:

$$r.s = \{\langle r_1, .., r_{n-1}, s_2, .., s_m\rangle \mid \langle r_1, .., r_n\rangle \in r \ \wedge \langle s_1, .., s_m\rangle \in s \ \wedge \ (r_n = s_1)\}$$

If r is a value expression ($n = 1$) and s is a field expression ($m = 2$), the expression $r.s$ gives the standard relational image of r under s. If r is a field expression ($n = 2$) and s is a value expression ($m = 1$), the expression $r.s$ gives those elements of the domain of r whose mapping under r equals some element in s.

A field expression gives a binary relation that can contain the initial value of a field (PreField), a Cartesian product of two value expressions, and the join, intersection, union, or difference of two field expressions. Furthermore, it can be the result of overriding a field expression with another field expression using the ++ operator:

$$r \texttt{ ++ } s = s \texttt{ + } (r \texttt{ - } \{\langle r_1, .., r_n\rangle : r \mid r_1 \in \mathit{domain}(s)\})$$

That is, (r ++ s) contains all the elements of s and those elements of r not mapped by s (The operators + and - give the set union and difference, respectively).

Our logic also includes the reflexive transitive closure of a binary relation, denoted by the * operator. For a homogeneous binary relation $r : T \rightarrow T$, the relation * is defined as follows:

$$\texttt{*}\ r = \mathit{iden} \texttt{ + } r \texttt{ + } r.r \texttt{ + } r.r.r \texttt{ + } ...$$

where *iden* is the identity relation. The expression $x.*r$ succinctly represents all the elements reachable from x via the r relation.

2.3 Examples

Figure 3 gives an implementation of a doubly linked list data structure. A list contains a head field that gives its first entry. Each entry has an integer data field[1] and links to its next and previous entries in the list.

Figures 4 - 7 show the summaries inferred by our technique for some small procedures manipulating linked lists. The first two examples illustrate the approximations of basic program statements, while the last two illustrate the accuracy of the summaries generated for some loops. In each summary, the final (primed) values of fields, variables, and types are given based on their initial (unprimed) values. The

[1]Our analysis does not currently support integer arithmetic, but it does support the use of int as a datatype.

```
class Entry {
  int data;
  Entry next;
  Entry prev;

  Entry(int d) {
    data = d;
    next = null;
    prev = null;
  }
}
class List { Entry head; }
```

Figure 3: A linked list declaration

value returned by a procedure proc is denoted by proc_return. Local variables of procedures do not appear in the generated summaries. Furthermore, we assume that all the constraints in a summary are implicitly conjoined.

```
/*
 aliasDemo_return = e2.(data ++ (e1 → 0))
 data' = data ++ (e1 → 0)
 head' = head
 next' = next
 prev' = prev
 Entry' = Entry
 List' = List
*/

int aliasDemo (Entry e1, Entry e2) {
   e1.data = 0;
   return e2.data;
}
```

Figure 4: Possible aliasing of list elements

The aliasDemo procedure shows the basics of our abstraction. An update to a field is encoded by a relational override which permits a succinct description of exactly which parts of a relation are changed and how. The constraint (data' = data ++ (e1 → 0)) encodes that the data field of e1 is set to 0 while the data field of other objects is unchanged. Navigations are encoded by relational joins. It should be noted that the generated summary specifies the behavior of the procedure in the general case; it is correct whether or not e1 and e2 are aliased.

In the next examples, we assume that any relation not listed in a summary has the same value in the pre- and post-states.

The insert procedure (Figure 5) allocates a new entry containing the given data d, and inserts it at the beginning of the receiver list, this. The summary uses New_Entry as a symbolic name for the allocated object. The constructor extends the data relation by a mapping from New_Entry to d, and extends next and prev relations by a mapping from New_Entry to null. The insert procedure also mutates the head field of the receiver list. Furthermore, it updates the next relation to include a mapping from New_Entry to this.head. This update overrides the update previously performed by the constructor. Lastly, based on whether this.head is null or not, the prev field of this.head may be changed to New_Entry. Since the mapping (this.head → New_Entry) may or may not be included in the final prev' relation, it is added to the upper bound of prev'. Because the final mapping of this.head under prev' is unknown, the lower bound contains no mapping for this.head.

```
/*
 Entry' = Entry + New_Entry
 data' = data ++ (New_Entry → d)
 head' = head ++ (this → New_Entry)
 next' = next ++ (New_Entry → this.head)
 prev' ⊆ (prev ++ New_Entry → null)
          + (this.head → New_Entry)
 prev' ⊇ (prev ++ New_Entry → null)-(this.head → Entry')
*/

void insert(int d) {
   Entry e = new Entry(d);
   Entry tmp = this.head;
   this.head = e;
   e.next = tmp;
   if (tmp != null)
      tmp.prev = e;
}
```

Figure 5: Inserting an element in a list

```
/*
 data' = data ++ ((this.head.*next & (Entry'-null)) → d)
*/

void init(int d) {
   Entry curr = this.head;
   while (curr != null) {
      curr.data = d;
      curr = curr.next;
   }
}
```

Figure 6: Initializing a list

The init procedure (Figure 6) assigns a given value to the data field of each entry of the receiver list. It involves a simple loop which matches the loop pattern optimized by our analysis: a linked data structure is traversed to the end. For such loops, the set of values that the loop variable takes in different iterations can be precisely specified by an expression. In this example, the loop variable is curr and the values it takes is specified by the expression (this.head.*next & (Entry' - null)), i.e. all non-null objects reachable from this.head by following the next link. This expression gives the exact set of objects whose data fields are mutated. In this example, all those fields get the value d which is constant with respect to the loop. Therefore, the summary generated in this case is the full specification of the procedure's behavior.

```
/*
 search_return ⊆ ((this.head.*next) & (null + data.d))
 search_return ⊇ ∅
*/

Entry search(int d) {
   Entry curr = this.head;
   while ((curr != null) && (curr.data != d))
      curr = curr.next;
   return curr;
}
```

Figure 7: Searching a list

The next example is the search procedure that returns the first entry in the receiver list whose data matches the given d, or null if no such entry exists. The procedure can be implemented in different ways. We use the implementation given in Figure 7 to illustrate how our analysis exploits loop conditions to produce more precise results even when the loop does not match the optimized pattern. The analysis infers that in each iteration, the curr variable points to some object reachable from this.head by following the next link, and thus its final value (the return value of the procedure) must be in this.head.*next. Since the final value is outside of the loop, it must violate the loop condition. That is, it must either be null or its data field must be equal to d. The set of all objects satisfying either of these conditions is encoded by the relational expression (null + data.d). Therefore, the return value belongs to the intersection of the above expressions, i.e. search_return ⊆ this.head.*next & (null + data.d)

3. ABSTRACTION

3.1 Definitions

Environments.

The set of concrete values of an object-oriented program comprises the values of variables, fields, and types. The concrete value of a variable is an object (which is denoted by a singleton set to provide a uniform definition), the concrete value of a field is a mapping from objects to objects, and the concrete value of a type is the set of all objects of that type. If Obj represents the set of all objects in a program, the set of all possible concrete values $CVal$ is defined as follows:

$$CVal = \mathcal{P}(Obj) \cup \mathcal{P}(Obj \times Obj)$$

where $\mathcal{P}(S)$ denotes the powerset of a set S.

The set of abstract values, $AVal$, comprises the three kinds of relational expressions defined in the previous section: value expressions, field expressions, and allocation expressions.

$$AVal = ValueExpr \cup FieldExpr \cup AllocExpr$$

A concrete state maps each variable, field, and type to a concrete value. We use C to denote the set of all possible well-typed concrete states of a program:

$$C = Var \cup\ Field \cup Type \rightarrow CVal$$

We abstract the set of all concrete states that may occur at a program point with a pair of environments $\langle E^u, E^o \rangle$. An environment is a mapping from each variable, field, and type to an abstract value:

$$Env = Var \cup Field \cup Type \rightarrow AVal$$

The environments E^u and E^o represent the under- and the over-approximation of the concrete states, respectively.

The set of concrete states at a program point is abstracted using the independent attribute method: an abstract environment maps each variable, field, and type to an abstraction of the set of its values in the given concrete states. We therefore lose the correlation between the values of different variables at a program point.

The meaning of the abstraction is given by the concretization function γ defined as follows:

$$\gamma : Env \times Env \rightarrow \mathcal{P}(C)$$
$$\gamma(\langle E^u, E^o \rangle) =$$
$$\{c \mid \exists c_0, \delta, \forall x,\ [\![E^u(x)]\!]^\delta_{c_0} \subseteq c(x) \subseteq [\![E^o(x)]\!]^\delta_{c_0}\}$$

where c_0 is a well-typed initial state, δ is a binding of allocated symbolic objects to concrete objects not used in c_0, and $[\![e]\!]^\delta_{c_0}$ denotes the meaning of a relational expression e under the bindings defined by c_0 and δ. The concretization function γ gives the set of all concrete states in which the values of all variables, fields, and types are under-approximated by E^u and over-approximated by E^o.

Lattice of Environments.

We define a partial order $\sqsubseteq$ over pairs of environments as follows:

$$\langle E^u_1, E^o_1 \rangle \sqsubseteq \langle E^u_2, E^o_2 \rangle \iff$$
$$\forall x,\ (E^u_1(x) \supseteq E^u_2(x)) \wedge (E^o_1(x) \subseteq E^o_2(x))$$

This partial order defines a pointed lattice over the set of all pairs of environments, with a top $\top = \langle E_\emptyset, E_{univ} \rangle$ and a bottom $\bot = \langle E_{univ}, E_\emptyset \rangle$ where $E_\emptyset$ represents the environment in which everything is mapped to the empty expression, and E_{univ} denotes the environment in which everything is mapped to the universe of all elements.

The lattice join $\sqcup$ is defined as follows:

$$\langle E^u_1, E^o_1 \rangle \sqcup \langle E^u_2, E^o_2 \rangle =$$
$$\langle\ \lambda x.E^u_1(x)\ \texttt{\&}\ E^u_2(x),\ \ \lambda x.\ E^o_1(x)\ \texttt{+}\ E^o_2(x)\ \rangle$$

It computes the least upper bound of two pairs of environments by intersecting the lower bounds of all variables, fields, and types in the two environments, and unioning their upper bounds.

We also define the following widening operation to stabilize infinitely ascending chains in the lattice.

$$\langle E^u_1, E^o_1 \rangle \bigtriangledown \langle E^u_2, E^o_2 \rangle =$$
$$\textbf{let } \langle E^u_3, E^o_3 \rangle = \langle E^u_1, E^o_1 \rangle \sqcup \langle E^u_2, E^o_2 \rangle \textbf{ in}$$
$$\langle\ \lambda v.\ \textbf{if } \exists x, r,\ (E^u_3(v) = x\ \texttt{\&}\ x.r\ \texttt{\&}\ ..\ \texttt{\&}\ x.r^{(k)})$$
$$\textbf{then } \emptyset \textbf{ elseif } |E^u_3(v)|\ \geq n$$
$$\textbf{then } \emptyset \textbf{ else } E^u_3(v)$$
$$\cup\ \lambda f.\ \textbf{if } |E^u_3(f)|\ \geq n$$
$$\textbf{then } \emptyset \textbf{ else } E^u_3(f)$$
$$\cup\ \lambda t.\ E^u_3(t),$$
$$\lambda v.\ \textbf{if } \exists x, r,\ (E^o_3(v) = x\ \texttt{+}\ x.r\ \texttt{+}\ ..\ \texttt{+}\ x.r^{(k)})$$
$$\textbf{then } x.*r \textbf{ elseif } |E^o_3(v)|\ \geq n$$
$$\textbf{then } type(v) \textbf{ else } E^o_3(v)$$
$$\cup\ \lambda f.\ \textbf{if } |E^o_3(f)|\ \geq n$$
$$\textbf{then } domainType(f) \rightarrow rangeType(f)$$
$$\textbf{else } E^o_3(f)$$
$$\cup\ \lambda t.\ \textbf{if } \exists l_1, .., l_m : SymObj,\ (E^o_3(t) = l_1\ \texttt{+}\ ..\ \texttt{+}\ l_m)$$
$$\textbf{then } E^o_3(t) + symObjSet(t)\ \rangle$$

It first computes the least upper bound of the given pairs of environments, then approximates the result based on the constant bounds k, m, and n with the following rules:

- A union of k joins $x\ \texttt{+}\ x.r\ \texttt{+}\ ..\ \texttt{+}\ x.r^{(k)}$ is overapproximated by $x.*r$, where $*r$ is the reflexive transitive closure of r. In order to apply this rule, r need not be a single field name; it can be any arbitrary relational expression. Similarly, an intersection of k joins is under-approximated by the empty set.
- If m objects are already allocated for a type, the widening operation generates a symbolic object set, of unspecified cardinality, to be used in future allocations.
- Any expression whose number of operators is greater than a limit n is over-approximated by a universal relation of the appropriate type, and under-approximated by the empty set.

Initial Abstraction.

Summarizing a procedure p starts by generating a pair of initial abstract environments $\langle E_0^u, E_0^o \rangle$ in which $E_0^u = E_0^o$ and each formal parameter, field, and type accessed in p is mapped to a symbolic name. Our analysis makes no assumptions about the possible aliasings between names, and thus the result is valid for all possible aliasings.

3.2 Transfer Functions

The transfer function

$$\bar{\mathcal{F}} : Stmt \rightarrow (Env \times Env) \rightarrow (Env \times Env)$$

models the effects of program statements on the abstract environments. The expressions generated by this function are syntactically simplified using the equivalence-preserving transformations described in Section 3.3.

Evaluation in an environment: The auxiliary function *eval* evaluates a program expression in a given environment:

$$\begin{array}{l} eval : PExpr \rightarrow Env \rightarrow ValueExpr \\ eval(c : Const, E) = \{\texttt{<}c\texttt{>}\} \\ eval(v : Var, E) = E(v) \\ eval(e.f, E) = eval(e, E).E(f) \end{array}$$

That is, the evaluation of a constant is always a unary relation corresponding to that constant. The evaluation of a program variable will be just a lookup in the environment, and the evaluation of a navigation expression is defined recursively.

Assignments to locals: Assigning an expression e to a local variable v is abstracted by the following rule:

$$\begin{array}{l} \bar{\mathcal{F}}(v \texttt{ := } e, \langle E^u, E^o \rangle) = \\ \quad \langle E^u[v \mapsto eval(e, E^u)], E^o[v \mapsto eval(e, E^o)] \rangle \end{array}$$

That is, the lower and upper bounds of e become the lower and upper bounds of x.

Field updates: Assigning an expression e_2 to a field f of the object described by an expression e_1 is abstracted by the following rule:

$$\begin{array}{l} \bar{\mathcal{F}}(e_1.f \texttt{ := } e_2, \langle E^u, E^o \rangle) = \\ \quad \textbf{let } x_1^u = eval(e_1, E^u), x_1^o = eval(e_1, E^o), \\ \qquad x_2^u = eval(e_2, E^u), x_2^o = eval(e_2, E^o) \textbf{ in} \\ \quad \textbf{if } (x_1^u = x_1^o) \wedge (x_2^u = x_2^o) \\ \quad \textbf{then } \langle E^u[f \mapsto E^u(f) \texttt{ ++ } x_1^u \rightarrow x_2^u], \\ \qquad E^o[f \mapsto E^o(f) \texttt{ ++ } x_1^o \rightarrow x_2^o] \rangle \\ \quad \textbf{else } \langle E^u[f \mapsto E^u(f) \texttt{ - } x_1^o \rightarrow rangeType(f)], \\ \qquad E^o[f \mapsto E^o(f) \texttt{ + } x_1^o \mapsto x_2^o] \rangle \end{array}$$

Two cases are distinguished:

- Strong update: if the lower bounds of e_1 and e_2 are syntactically the same as their upper bounds, then the computed values for both expressions are exact. In this case, the value of e_2 overrides the previous value of $e_1.f$.
- Weak update: if the values computed for e_1 and e_2 are not necessarily exact, it is not clear for which object the f field was mutated, or how it was mutated. Therefore, the over-approximation allows the f field of any of the objects represented by the upper bound of e_1 to be mapped to any of the objects represented by the upper bound of e_2. The lower bound leaves all possible mutated objects unconstrained.

Allocations: We assume that an allocation statement `v = new t(`e_1, .., e_n`)` is broken into two consecutive statements[2]: `v = new t; v.init(`e_1, .., e_n`);`. The first statement does the actual allocation, and the second one calls the proper constructor on the allocated object. Here, we explain the abstraction of the allocation statement. The call to the constructor will be abstracted similar to the other method calls (explained later).

$$\begin{array}{l} \bar{\mathcal{F}}(v \texttt{ := new } t, \langle E^u, E^o \rangle) = \\ \quad \textbf{let } s = getSymObjSet(t) \textbf{ in} \\ \quad\quad \textbf{if } s \neq \emptyset \textbf{ then } \langle E^u[v \mapsto \emptyset], E^o[v \mapsto s] \rangle \\ \quad\quad \textbf{else let } l = symObj(t) \textbf{ in} \\ \quad\quad\quad \langle E^u[t \mapsto E^u(t) \texttt{ + } l, v \mapsto l], \\ \quad\quad\quad\ E^o[t \mapsto E^o(t) \texttt{ + } l, v \mapsto l] \rangle \end{array}$$

The *getSymObjSet* function returns the symbolic object set generated for a type t, if it exists, and the *symObj* function generates a fresh symbolic object of a type t.

Allocating an object of type t is abstracted by distinguishing two cases:

- If as a result of a previous widening, a symbolic object set is already generated for a type t, meaning that we have already hit the limit for enumerating new objects, the new object is over-approximated by that same symbolic set and under-approximated by the empty set. Since the cardinality of the symbolic set is unconstrained, no other updates are necessary.
- If no symbolic set has been generated, we generate a fresh symbol as the exact value of the new object. This symbol will then be added to the allocation expression of t.

Call sites: During the analysis, different calls to a procedure p may have different summaries. This is because the abstractions of field updates and allocation statements are based on whether the values computed for variables, fields, and types are exact or not. However, it is often the case that the summaries are structurally identical, and we can avoid re-computing them from scratch at each call site. One can predict whether or not two summaries will be structurally identical by looking at their *context*, an abstraction that records the parts of the environments with exact values:

$$\begin{array}{l} context(\langle E^u, E^o \rangle) = \\ \quad \{v : var \mid E^u(v) = E^o(v)\} \cup \\ \quad \{f : field \mid E^u(f) = E^o(f)\} \cup \\ \quad \{t : type \mid getSymObjSet(t) = \emptyset\} \end{array}$$

Given a context of p, our technique generates a *summary template* which can then be instantiated to produce a summary for a call to p.

Consider a call to a procedure p in an environment pair $\langle E^u, E^o \rangle$ with context c. We compute a summary template t from c by first generating a pair of environments whose context is the same as c, but maps all variables, fields, and types to fresh symbolic values, and then summarizing the

[2]This may be done using the Soot framework [17].

body of p on that generated pair of environments:

$$
\begin{array}{l}
template(p, c) = \\
\quad \textbf{let } E_1^u = \lambda v.\ sym(v) \ \cup\ \lambda f.\ sym(f) \ \cup\ \lambda t.\ sym(t), \\
\qquad E_1^o = \ \lambda v.\ \textbf{if } (v \in c) \textbf{ then } E_1^u(v) \textbf{ else } sym(v) \\
\qquad\quad \cup\ \lambda f.\ \textbf{if } (f \in c) \textbf{ then } E_1^u(f) \textbf{ else } sym(f) \\
\qquad\quad \cup\ \lambda t.\ \textbf{if } (t \in c) \textbf{ then } E_1^u(t) \\
\qquad\qquad\qquad \textbf{else } E_1^u(t) \texttt{ + } getSymObjSet(t) \\
\quad \textbf{in } \bar{\mathcal{F}}(p\_\texttt{body}, \langle E_1^u, E_1^o \rangle)
\end{array}
$$

where $sym(x)$ generates a fresh symbolic value for x.

The summary of the call site is then computed by instantiating t with the values from $\langle E^u, E^o \rangle$. That is, by substituting the values of actual parameters, and fields at the call site for the symbols used in the summary template.

$$
\begin{array}{l}
\bar{\mathcal{F}}(\texttt{proc}(e_1, ..., e_n), \langle E^u, E^o \rangle) = instantiate(\\
\quad template(\texttt{proc}, context(\langle E^u, E^o \rangle)), \langle E^u, E^o \rangle)
\end{array}
$$

The summary template t is saved and associated with the context c. If another call to p is later encountered with the same context c, then its summary can be computed by instantiating t with the values from its environment, rather than computing the summary directly from scratch.

If a procedure p has x formal parameters, accesses y different fields, and allocates z different types, then it has 2^{x+y+z} possible contexts. It should be noted that the empty context represents the case where none of the computed values are exact. Therefore, the template summary generated for the empty context only performs weak updates and although it is not precise, it can be used in all other contexts too. To balance between precision and performance, we summarize each procedure on demand for its first l distinct contexts where l is a constant. Any further calls whose corresponding contexts are not visited before will be replaced by the procedure's template summary for the empty context.

Return statements: A procedure named proc is allocated a special variable proc_return to hold its return value. A return statement is simply an assignment of a value to that variable.

$$
\begin{array}{l}
\bar{\mathcal{F}}(\textbf{return } e, \langle E^u, E^o \rangle) = \\
\quad \langle\ E^u[\texttt{proc\_return} \mapsto\ (E^u(\texttt{proc\_return}) \texttt{ \& } eval(e, E^u))], \\
\quad\ E^o[\texttt{proc\_return} \mapsto\ (E^o(\texttt{proc\_return}) \texttt{ + } eval(e, E^o))]\ \rangle
\end{array}
$$

If the procedure has multiple return statements, the return values accumulate in E^u and E^o by intersecting the lower bounds and unioning the upper bounds.

Branches: Conditional statements are abstracted by first abstracting each branch independently, then combining the results using the lattice least upper bound operator.

$$
\begin{array}{l}
\bar{\mathcal{F}}(\textbf{if } (c)\ S_1 \textbf{ else } S_2, \langle E^u, E^o \rangle) = \\
\quad \bar{\mathcal{F}}(S_1, \langle E^u, E^o \rangle) \sqcup \bar{\mathcal{F}}(S_2, \langle E^u, E^o \rangle)
\end{array}
$$

Loops: Loops are abstracted by successively abstracting the body and joining each new abstraction with the abstraction of previous iterations. To produce more precise summaries, we intersect the loop condition with the variables used in it at the beginning of each iteration. Furthermore, the termination condition (negation of loop condition) is intersected with the final value of the variables in the condition.

$$
\begin{array}{l}
\bar{\mathcal{F}}(\textbf{while}(c)\ S,\ \langle E^u, E^o \rangle) = \\
\quad \textbf{let } \langle E_1^u, E_1^o \rangle = \textbf{Fix } \langle E^u, E^o \rangle. \\
\qquad \bar{\mathcal{F}}(S, \langle addCond(c, E^u), addCond(c, E^o) \rangle) \bigtriangledown \langle E^u, E^o \rangle \\
\quad \textbf{in } \langle addCond(\neg c, E_1^u), addCond(\neg c, E_1^o) \rangle
\end{array}
$$

where $\bigtriangledown$ is the lattice widening operator, and $addCond(c, E)$ is a function that intersects a relational encoding of a condition c with the values of c's variables in an environment E:

$$
\begin{array}{l}
addCond(v.f_1.f_2 \ldots f_n \texttt{ == } e, E) = \\
\quad \textbf{let } x = eval(v, E), y = eval(e, E) \\
\quad \textbf{in } E[v \mapsto x \texttt{ \& } E(f_1).E(f_2) \ldots E(f_n).y]
\end{array}
$$

$$
\begin{array}{l}
addCond(v.f_1.f_2 \ldots f_n \texttt{ != } e, E) = \\
\quad \textbf{let } x = eval(v, E), y = type(e) \texttt{ - } eval(e, E) \\
\quad \textbf{in } E[v \mapsto x \texttt{ \& } E(f_1).E(f_2) \ldots E(f_n).y]
\end{array}
$$

$$addCond(c_1 \texttt{ \&\& } c_2) = addCond(c_1) \cap addCond(c_2)$$

$$addCond(c_1 \texttt{ || } c_2) = addCond(c_1) \cup addCond(c_2)$$

$$
\begin{array}{l}
E_1 \cup E_2 = \ \lambda x.\ E_1(x) \texttt{ + } E_2(x) \\
E_1 \cap E_2 = \ \lambda x.\ E_1(x) \texttt{ \& } E_2(x)
\end{array}
$$

For example, consider a loop accessing variables c and d whose values at the beginning of the loop are approximated by expressions c_0 and d_0, respectively. If the loop condition is ($c.f$==d), then the subset of c_0 that passes the condition belongs to $f.d_0$, the set of elements whose mapping under f is an element of d_0. The set of values that pass the loop condition is therefore (c_0 & $f.d_0$). The environment obtained by applying the *addCond* function maps c to this more accurate expression.

3.3 Simplifications

Expressions generated by the transfer function can often be simplified using relational logic equivalence rules, which reduce the size of an expression without changing its semantics. A subset of those rules is given in Figure 8.

Figure 8(a) gives a set of equivalence rules in relational logic that simplifies union, intersection, difference, Cartesian product, and composition of relations.

The rules in Figure 8(b) simplify relational expressions based on the semantics of reflexive transitive closure and relational override. The expression $*r$ contains all elements reachable by traversing r any number of times, and that r ++ $(x \rightarrow y)$ overrides any previous binding of $x.r$ with y.

The rules in Figure 8(c) are based on the semantics of types and allocations. Their validity is based on the fact that newly allocated objects do not alias objects in the pre-state. We write x_{new} and x_{pre} to distinguish the case that x is an allocated symbolic object from the case where it is a relation in the pre-state. A name with no subscript may be either.

3.4 Properties

3.4.1 Safety

The abstraction described above is *safe*, meaning that the generated under- and over-approximations account for all executions of the summarized procedure.

Figure 9 shows the relation between the concrete states c_i and abstract environments $\langle E_i^u, E_i^o \rangle$ for a safe abstraction. In this figure, γ is the concretization function, $\mathcal{F}$ is the concrete state transition function which can be defined by the operational semantics of the program statements, and $\bar{\mathcal{F}}$ is the abstract state transition function (the transfer function defined in Section 3.2).

In the interest of space, we just give a sketch of the safety proof. In order to prove safety, it is sufficient to show:

$x \texttt{ + } x = x$
$x \texttt{ - } (x \texttt{ - } y) = x \texttt{ \& } y$
$x.(x \to y) = y$
$x.r \texttt{ + } x.s = x.(r \texttt{ + } s)$
$x.r \texttt{ + } y.r = (x \texttt{ + } y).r$
$(x \texttt{ \& } c) \texttt{ + } (y \texttt{ \& } c) = (x \texttt{ + } y) \texttt{ \& } c$
$(x \to z) \texttt{ + } (y \to z) = (x \texttt{ + } y) \to z$

(a)

$x.*r.*r = x.*r$
$x \texttt{ + } (x.*r) = x.*r$
$(x \texttt{ + } x.r).*r = x.*r$
$r \texttt{ ++ } (x \to y) \texttt{ ++ } (x \to z) = r \texttt{ ++ } (x \to z)$
$r \texttt{ + } (r \texttt{ ++ } (x \to y)) = r \texttt{ + } x \to y$
$r \texttt{ \& } (r \texttt{ ++ } (x \to y)) = r \texttt{ - } (x \to rangeType(r))$

(b)

$type(x).(x \to y) = y$
$x.(type(x) \to y) = y$
$x \texttt{ \& } type(x) = x$
$r_{pre} \texttt{ - } (x_{new} \to y) = r_{pre}$
$x_{pre}.(r_{pre} \texttt{ ++ } (y_{pre} \to z) \texttt{ ++ } (t_{new} \to w)) =$
$\quad x_{pre}.(r_{pre} \texttt{ ++ } (y_{pre} \to z))$
$t_{new}.(r_{pre} \texttt{ ++ } (y_{pre} \to z) \texttt{ ++ } (t_{new} \to w)) = w$

(c)

Figure 8: A sampling of the simplification rules used. The function $type(e)$ denotes the set of all objects with the same type as an expression e.

1. The initial abstraction is safe. That is, $\forall c : C \mid c \in \gamma(\langle E_0^u, E_0^o \rangle)$
2. The composition property holds. That is, $\mathcal{F} \circ \gamma \subseteq \gamma \circ \bar{\mathcal{F}}$

The first property is valid because the symbolic names used in the pre-state are uninterpreted, meaning that they can be instantiated by any concrete values. The second property can be proved by cases: it holds for each program statement separately based on the definition of the corresponding transfer function and the operational semantics of that statement.

3.4.2 Termination

In order to prove that abstraction always terminates, it is sufficient to show that the abstraction of all loops and recursive procedures reaches a fixpoint after a finite number of iterations.

Both least upper bound and widening operations result in a lattice point which is either the same as one of their arguments or one that is higher than both of them. During the abstraction of a loop, if the join operation results in a lattice point which is the same as one of its arguments, then it has reached a fixpoint and the abstraction terminates. If the join instead results in a node higher in the lattice, the abstraction can make only a finite number of moves up the lattice before the abstract environment is widened to $\top$. This is because the length of all expressions is bounded by a constant size, and thus the height of the lattice is finite. Therefore, after a finite number of steps, the abstraction will either reach a fixpoint at $\top$ or a fixpoint at some lower point.

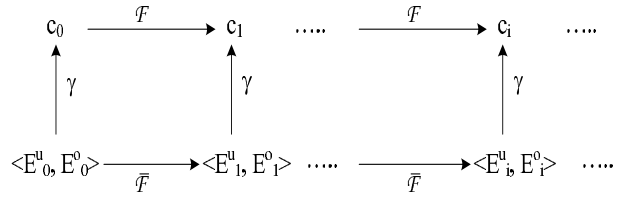

Figure 9: Relation between concrete states and abstract states

3.5 Optimization

A common loop pattern in heap-manipulating programs is iterating over a linked data structure: a loop traverses over a data structure using a variable x that acts as a cursor. In each iteration of the loop, the cursor variable is updated by taking one step along a relation r; the termination condition is that no further step along r can be taken. In a linked list, for example, x is a reference to a list entry, r is the *next* field from entry to entry, and the termination condition is $x.next$`!=`$null$. In a traversal using an iterator, x is the reference to the iterator, r is the specification field associated with iteration, and the termination condition is $hasNext(x)$.

The general form of the loop is:

```
while (cond(x, r)) {
   S1; x=next(x); S2; }
```

which allows arbitrary statements before and after the update of the cursor, so long as they do not mutate the relation r or assign to the cursor x.

For any loop in this form, our analysis treats the approximation of x by $x_0.*r$ (where x_0 is the initial value of x and $*r$ is the reflexive-transitive closure of r) as the exact set of values taken by x during the execution of the loop.

Having inferred the exact value of the loop variable, the analysis often generates more precise abstraction of the loop body by performing an additional optimization pass over the loop. This pass infers a more precise final value for any field which is (1) updated exactly once in the loop body and (2) the updating statement is of the form $e_1.f = e_2$ where e_1 has an exact value and e_2 is constant with respect to the loop. This additional precision will often carry through the abstraction rules. Recall the loop we saw in Figure 6:

```
void init(int d) {
  Entry curr = this.head;
  while (curr != null) {
    curr.data = d;
    curr = curr.next;
  }
}
```

It matches the optimized pattern, so the exact value of `curr` will be `this.head.*next`. After intersecting with loop condition, it will be `this.head.*next & (Entry' - null)`. Since the value of `curr` is exact, the abstraction of the statement `curr.data = d` is also exact, giving us the following:

```
data' = data ++
  ((this.head.*next & (Entry' - null)) → d)
```

Without this optimization we would have generated the following:

```
data' ⊇ data -
  ((this.head.*next & (Entry' - null)) → int)
data' ⊆ data +
  ((this.head.*next & (Entry' - null)) → d)
```

which allows an arbitrary subset of this.head.*next.data be overridden by d, and thus, is a much weaker specification.

4. EXPERIMENTS

We ran our technique to generate summaries for procedures from two Java libraries: the standard Java linked list implementation and the OpenJGraph package[9]. For each library, we summarize all procedures written in our supported subset of Java. The accuracy of summaries varied, but no summary took more than 3 seconds to generate even though some had as many as 81 procedure calls.

To evaluate the accuracy of the generated summaries, we used them to check properties of their corresponding procedures. We performed two experiments: (1) Summarize small, structurally complex procedures and compare those summaries to pre-existing full specifications. For this study, we used the Java linked list library. (2) Summarize procedures with more typical object-oriented code which uses several different data structures and makes many external calls. Since full specifications were not available, we assessed the summaries against manually written partial specifications. For this study, we used the OpenJGraph package.

The summaries were analyzed using the Alloy Analyzer [6]: for a procedure p and a specification S we checked if the formula $(summary(p) \implies S)$ holds, where $summary(p)$ is the summary generated by our technique. Since the Alloy Analyzer analyzes formulae in finite scopes, we first checked the summaries in a high scope and then inspected them manually. Our technique generated full specifications for 13 of the 30 procedures. In the remaining 17 procedures, our technique generated all frame conditions, and in 16 of them it inferred some major properties. In one of the procedures the generated summary was too rough to be useful for more than its frame conditions.

The experiments were done using the following constants: Maximum number of operators allowed in an expression before it is widened to the universal relation (n) = 1300. Maximum number of allocations enumerated before a set of objects is allocated (m) = 5. Maximum number of unions before widening to closure (k) = 3. Maximum number of contexts for which a procedure is summarized (l) = 5.

4.1 Java Linked List Implementation

We generated summaries for 10 procedures in Sun's standard implementation of linked list in the Java Collections Framework. The implementation represents a list as a circular doubly linked list with a dummy header entry. An integer field size keeps track of the size of the list. Since our current technique does not handle arithmetic expressions, we ignored all accesses to size, and only analyzed procedures that do not depend on integer arithmetic.

We generated summaries for 10 procedures: add, addFirst, addLast, remove, removeFirst, removeLast, getFirst, getLast, clear, and clone. These procedures are small and mostly self-contained, but their correctness depends on their complex mutations of the underlying doubly linked data structure.

```
/*
 clone_return = New_LinkedList
 element' ⊇ element
 element' ⊆ element + (set_New_Entry →
  (this.header.next.*next & (Entry'- this.header)).element)
 next' ⊇ next
 next' ⊆ next + (set_New_Entry → set_New_Entry)
 previous' ⊇ previous
 previous' ⊆ previous + (set_New_Entry → set_New_Entry)
 header' = header ++ (New_LinkedList → set_New_Entry)
 Entry' ⊇ Entry
 Entry' ⊆ Entry + set_New_Entry
 LinkedList' = LinkedList + New_LinkedList
*/

public Object clone() {
  LinkedList clone = new LinkedList();
  for (Entry e = header.next; e != header; e = e.next) {
    Entry newEntry = new Entry(e.element,
      clone.header, clone.header.previous);
    newEntry.previous.next = newEntry;
    newEntry.next.previous = newEntry;
  }
  return clone;
}
```

Figure 10: Clone

For each procedure, we checked if the generated summary is accurate enough to show that the procedure's JML specification holds. In 8 of the 10 analyzed procedures, we were able to check the complete specifications. The summaries generated for clone and remove were not accurate enough to prove the full specifications, although they still provided limited descriptions. These methods cause our technique difficulty because they mutate the same relation that is being traversed in a loop.

The summary generated for the clone method, shown in Figure 10, specifies that a fresh list is constructed and returned, and that it contains some freshly allocated entries whose elements are chosen from the set of the elements of the receiver list. Although the summary does not specify that the size of the constructed list is the same as the receiver list nor that the copy preserves order, it still specifies useful information about the procedure – for example, that the elements of the returned list are all chosen from the receiver list, and that the receiver list is not changed at all.

The remove method removes the first occurrence of a given element from the list. Although the list is updated at most once (for the first occurrence), the update is done as part of the loop. This prevents the analysis of the loop from stabilizing by inferring the closure, and the loop analysis instead terminates by widening to the universal relation. That is, the generated summary allows the values of the next and previous fields of any Entry object to change, meaning that any number of entries may be removed from the list. The summary therefore only provides the frame conditions.

4.2 OpenJGraph API

We used our tool to generate summaries for 20 procedures of the OpenJGraph package, an open source Java implementation of a graph package [9]. It represents a graph as a map from each vertex to a list of adjacent edges. A separate set records all the edges in the graph.

Instead of generating summaries for Java map and set using their implementations, we provided their specifications

Table 1: Partial specifications checked in graph procedures

Procedure	#Calls	Time (Sec)	Property
containsVertex(v)	1	0.1	no object fields are mutated
containsEdge(e)	1	0.1	no object fields are mutated
add(v)	11	0.1	v's final edge list is either empty or unchanged
			the edge lists of non-v vertices are unchanged
addEdge(e)	24	0.7	no new edges are added apart from e
			the vertex adjacency list is unchanged apart from the ends of e
addEdge(v1, v2)	29	0.8	the adjacency list of vertices is unchanged apart from v1 and v2
			no edge is added except possible one connecting v1 to v2
			the final graph has an edge connecting v1 to v2
removeEdge(e)	13	0.5	the vertex adjacency list is unchanged apart from the ends of e
			the final graph contains all edges of the original graph except e
remove(v)	27	0.9	the final graph does not contain v
minimumSpanningTree()	81	2.7	all edges of MST belong to the original graph
			nodes with no adjacent edges are not added to MST
			the set of edges of the original graph is unchanged

in the style of the summaries that our technique generates. Doing so demonstrates how our technique is compositional and can exploit off-the-shelf specifications when available.

Most methods in this package make a considerable number of external calls, either directly or indirectly. Our technique generates full specifications for 5 of the 20 procedures. In order to evaluate the quality of the summaries generated for the other 15 procedures, we checked them against two sets of properties:

- *Representation Invariants.* The graph package assumes some invariants about the shape of the data structures used and the consistency of the data stored in them. The invariants are given as informal comments in English. We encoded them as relational invariants in Alloy and checked whether or not our generated summaries are sufficient to show the preservation of these invariants. 6 invariants were checked in 15 procedures. Out of those 90 checks, 67 of them passed and the other 23 failed. The invariant with the highest failure rate is the one stating that the edge-set data structure contains all edges in the adjacency lists of the vertices. In most of the procedures, the adjacency lists and edge set are modified within branches. Since the effects of different branches are unioned to form the final summaries, the summaries for such procedures allow any combination of modifications, and thus they are too weak to show the edge-set consistency invariant.

- *Post Conditions.* Lacking formal specifications of the procedures, we checked the generated summaries for the 15 procedures against some partial post conditions. Table 1 shows a subset of the properties that the summaries were sufficient to check. The *#Calls* column gives the number of procedure calls made by the analyzed procedure and its callees. We only counted the number of distinct method calls; calls made to the same procedure in a loop are counted only once. The *Time* column gives the time to generate a summary using a Pentium 4, 1.8GHz with 512 MB of RAM. The time is given in seconds. The *Property* column is an English description of the checked partial post-condition.

The most important cases of information loss are from the following two cases:

(1) A procedure which returns a boolean value, as happens in containsEdge and containsVertex. Our technique unions the possible return values and reports that the return value is either true or false. The summary still provides a frame condition indicating which variables and fields are unchanged by the procedure, but the information about the return value is not useful. The two such cases in this study have no side effects, so the summaries are the full frame conditions.

(2) A procedure with a loop that mutates the same link it is traversing, as in remove. Our technique is able to determine whether or not the resulting elements are a subset of the prior elements, but it typically cannot tell which will be removed or if the order will remain the same.

5. RELATED WORK

The problem of detecting side effects of a heap manipulating procedure has been widely studied. Side effect analyses [2, 5, 8, 11, 14] compute the set of heap locations that may be mutated by a procedure, typically by using a pointer analysis to approximate the set of objects to which a reference variable could point.

Our method seeks the same end by different means: it uses relational expressions to approximate the set of objects pointed to by a variable, and then uses them to safely identify the set of locations that may be mutated by a procedure. Our method differs from side-effect analyses in that it computes both *may* and *must* side-effects to generate more precise summaries. Furthermore, our summaries specify constraints not only on *what* locations can be mutated by a method but also on *how* those locations are mutated.

Tkachuk and Dwyer use side effect analysis to generate summaries [16]. To check a property of a program unit more efficiently, they approximate the behavior of the rest of the program with summaries. Similar to our approach, they generate both may and must side effects and specify the mutations that can be performed by a procedure. However, our analysis is capable of generating more precise summaries (1)

by recording the history of field updates by symbolic override expressions, rather than conservatively updating the field in all objects that may be aliased, and (2) by exploiting loop conditions in abstracting loop bodies.

Shape analysis techniques [1, 7, 10, 12, 13, 18] also compute an abstraction of code and use it to verify some properties of linked data structures. They encode the heap as a graph in which the nodes represent objects and edges represent field relations. Since the graph can be arbitrarily large, they abstract it by merging all nodes equivalent under a particular set of shape properties into a single abstract node. Although the resulting abstract domain is finite, it is still very large, and thus the approximating algorithms have high complexity. Furthermore, the precision of their abstraction depends on the set of predicates of interest provided by the user. In contrast, our technique requires no user-provided annotations and provides a very lightweight, fast approximation that can later be refined on demand using a fully automatic framework similar to [15].

6. CONCLUSIONS

We have described a lightweight, flow-sensitive, context-sensitive technique for automatically generating symbolic summaries of object-oriented procedures. These summaries, expressed in a relational logic with transitive closure, can be thought of as detailed frame conditions; they describe which memory locations might be changed and in what ways.

This technique was originally conceived as a means of generating first approximations of procedure specifications, to be later refined if needed. To that end, it is focused on generating summaries that are safe (they describe a superset of possible program behaviors), fast to generate (the time to extract a summary is linear in the size of the procedure), and small (we are often able to concisely summarize the effect of a loop by using transitive closure).

We evaluated our technique using 30 heap-manipulating procedures from two Java libraries. Our current experiments suggest that this analysis represents a useful balance between tractability and accuracy. More experiments have yet to be done to evaluate the accuracy of the generated summaries for larger programs.

We have found that matching a common loop pattern (simple linked-data traversal) generates considerably more precise summaries at a very low cost. We expect that several more common but simple patterns could be similarly beneficial. One such pattern we are considering is for remove operations; their summaries are currently not very accurate, but they often have short, precise, relational specifications.

Currently, branch points always introduce imprecision into our summaries since we need to account for arbitrarily complex conditions. However, many conditionals have simple conditions and can be precisely summarized with a relational expression, and there is no fundamental reason why our technique cannot do so. Such an extension might permit us to produce precise summaries for simple conditional procedures such as `isEmpty` methods.

While we produce correct summaries in the presence of aliasing, the addition of some lightweight alias analysis should enable us to produce more concise summaries in several common cases. Our technique might also benefit from an escape analysis, to help establish which mutations are actually visible to the caller.

Acknowledgments

We are grateful to Alexandru Salcianu, Viktor Kuncak, and Derek Rayside for their advice and useful discussions.

7. REFERENCES

[1] I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. *In Proc. of VMCAI*, 2005.

[2] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *In Proc. of POPL*, 1993.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proc. of POPL*, pages 238–252, Jan 1977.

[4] T. Freeman and F. Pfenning. Refinement types for ML. *In Proc. of Programming Languages Design and Implementation*, 1991.

[5] M. Hind and A. Pioli. Which pointer analysis should I use? *In Proc. of International Symposium on Software Testing and Analysis*, 2000.

[6] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. *In Proc. of Foundations of Software Engineering*, pages 62–73, Sep 2001.

[7] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. *In Proc. of SAS*, pages 246–264, 2004.

[8] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. *In Proc. of International Symposium on Software Testing and Analysis*, 2002.

[9] OpenJGraph. http://openjgraph.sourceforge.net/.

[10] A. Podelski and T. Wies. Boolean heaps. *In Proc. of Static Analysis Symposium*, 2005.

[11] A. Rountev. Precise identification of side-effect-free methods in java. *IEEE International Conference on Software Maintenance*, 2004.

[12] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[13] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[14] A. D. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. *In Proc. of VMCAI*, 2005.

[15] M. Taghdiri. Inferring specifications to detect errors in code. *In Proc. of Automated Software Engineering*, pages 144–153, Sep 2004.

[16] O. Tkachuk and M. Dwyer. Adapting side effects analysis for modular program model checking. *In Proc. of Foundations of Software Engineering*, 2003.

[17] R. Valle-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. *In Proc. of CASCON*, 1999.

[18] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. On field constraint analysis. *In Proc. of VMCAI*, 2006.