Finding Bugs with a Constraint Solver

Daniel Jackson & Mandana Vaziri
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139
{dnj, vaziri}@lcs.mit.edu

ABSTRACT

A method for finding bugs in code is presented. For given small numbers *j* and *k*, the code of a procedure is translated into a relational formula whose models represent all execution traces that involve at most *j* heap cells and *k* loop iterations. This formula is conjoined with the negation of the procedure's specification. The models of the resulting formula, obtained using a constraint solver, are counterexamples: executions of the code that violate the specification.

The method can analyze millions of executions in seconds, and thus rapidly expose quite subtle flaws. It can accommodate calls to procedures for which specifications but no code is available. A range of standard properties (such as absence of null pointer dereferences) can also be easily checked, using predefined specifications.

KEYWORDS

Detecting bugs; relational formulas; Alloy language; constraint solvers; testing; static analysis; model checking.

1 INTRODUCTION

This paper describes a method for finding bugs in code. It requires no user input beyond a specification, is completely automatic, and, when a bug is detected, provides a counterexample trace through the code. It uses semantic technology in the style of model checking, and typically considers huge numbers of possible executions. It can thus discover quite subtle bugs.

ISSTA'00, Portland, Oregon.

The specification language is Alloy [22], an object modeling language that has been used primarily for describing high level specifications and designs. Alloy is a small and succinct language, and since it subsumes first order logic with transitive closure, can express a wide range of properties.

The general problem of determining whether a program meets its specification is undecidable, so automation can only be obtained in return for some compromise. In our case, we want to detect as many errors as possible without generating spurious error reports, so we have compromised completeness. Errors may be missed, but reported errors correspond to real traces.

Traditional shape analyses are directed at compiler optimization, and therefore make a different compromise. If the analysis infers a property of the code, it must hold, for otherwise an unsound optimization may be performed. When a property is not inferred, however, it may still hold. Put another way, in checking a property of the code, traditional analyses answers 'yes' or 'maybe'; our analysis answers 'no' or 'maybe'.

Whether an entire class of results – bugs reported or checks passed – can be trusted is not the only issue. *Accuracy* matters too: namely how often an answer, or an inference drawn from its omission, is correct. A bug detection scheme that guarantees answers that never arise in practice is less useful than one that offers no guarantees, but which nevertheless finds most bugs and gives few spurious reports.

For the properties our analysis addresses, we believe that conservative schemes are likely to be less accurate. Although extremely effective for simple properties (such as type checking), conservative analyses degrade as properties become more complex. In the context of checking abstract designs involving – like the code we consider here – unbounded data structures, we have found that an analysis based on considering finite instances exposes many errors [19], while generic abstractions that would allow 'proof' of properties are hard to find.

In the following sections, we show some sample results from our analysis (Section 2), describe its workings (Section 3) and report on its application to a benchmark suite of small

list processing procedures (Section 4). Throughout, our focus is on the extraction of a relational formula from the code and specification; our method for solving the formula is explained elsewhere [24].

2 ILLUSTRATION

Figure 1 shows the code of an (incorrect) procedure for deleting an element from a singly linked list. Below the procedure, we give a variety of partial specifications, to illustrate the expressive power of our specification language, and the kinds of specification our method accommodates. In practice, one would probably give only specification (3).

The fields of the *List* class are treated as binary relations, as if they had been declared as

$$next : List \rightarrow List$$
$$val : List \rightarrow Val$$

Primed names represent the value of a relation after execution; unprimed names represent the value before. The plus and star symbols denote transitive and reflexive transitive closure, so *l.*next'*, for example, represents the set of list cells reachable from the argument *l* after execution. The specifications say that (1) the set of cells reachable after is a subset of those reachable before – ie, that no cells are added by the procedure; (2) that no cell reachable after has a value field equal to *v* after; (3) that the set of cells reachable after is equal to the set reachable before, minus the set of cells with the value *v* before; (4) that no heap cell suffers a change in value; and (5) that if the list is acyclic before (that is, no cell *c* reachable from *l* in zero or more steps reaches itself in one or more steps), then it is acyclic after.

Specifications (1), (4) and (5) hold; (2) and (3) do not. The result of running our analysis for specification (2) is shown in Figure 2. This is the output produced by our constraint analyzer, and is thus not tailored to the analysis of code. It could obviously be improved; the path, for example, might be shown as a list of program statements.

The initial values of the arguments (*l* and *v*) and the fields (*next* and *val*) are given: *l* points to a list with a single cell, and a non-null value field. The variables of the form *Eij* give the edges of the control flow graph (Figure 5) that are traversed: *E01*, for example, represents the edge from node 0 to node 1 (and the execution of the statement *prev = NULL*). Intermediate values are given of both the program variables (*prev1*, for example, denoting the value of *prev* after the first assignment), and the fields (*next1*, for example, denoting the value of *next* after the field-setting statement). The final values of the fields, *next'* and *val'*, are in this case identical to the initial values. The skolem constant *c* is the witness to the violation of the quantifier in the specification: it shows that *L0* is the cell that should have been removed.

This counterexample exposes a bug: that the first list cell

```
class List {List next; Val val;}
class Procedures {
…
void static delete (List l, Val v) {
  List prev = null;
  while (l != NULL)
    if (l.val == v) {
      prev.next = l.next ;
      return; }
    else {
      prev = l ;
      l = l.next ;
  }
…
// 1: no cells added
l.*next' in l.*next
// 2: no cell with value v after
no c: l.*next' | c.val' = v
// 3: cells with value v removed
 l.*next' = l.*next – {c | c.val = v}
// 4: no cells mutated
all c | c.val = c.val'
// 5: no cycles introduced
no c: l.*next | c in c.+next -> no c: l.*next' | c in c.+next'
```

Figure 1: Sample procedure and specifications

cannot be deleted. To investigate further, we might add as a precondition that *v* not occur in the first cell:

$$l.val \mathrel{!}= v$$

Running the analysis again, we obtain a new counterexample: a list with three cells, in which the last two cells share a value equal to *v*. This exposes a second bug: that the procedure only removes the first cell with a matching value. We might now record as a representation invariant that the list be free of duplicates:

$$all\ x |\ sole\ cell: l.*next \mid cell.val = x$$

(namely, that for every value *x* there is at most one cell reachable from *l* with that value). Repeating the analysis now finds no counterexamples.

The key observation underlying our method is that even though neither the flaws nor these specifications are trivial, counterexamples requires few objects: one list cell for the first bug and three for the second. Our conjecture is that it is feasible to analyze exhaustively all possible executions that involve no more than some small number of objects and a couple of loop iterations, and that such an analysis is likely to find many errors in practice.

Our constraint analyzer allows the user to set a *scope* that bounds the number of atoms of each type. The cost of analysis grows super-exponentially with scope, but we have found

```
Domains:
  List = {L0}
  Val = {V0}
Sets:
  E01 = traversed
  E12 = (null)
  E13 = traversed
  E34 = traversed
  E45 = traversed
  E52 = traversed
  E36 = (null)
  E67 = (null)
  E78 = (null)
  E82 = (null)
  I = L0
  I1 = L0
  prev = L0
  prev1 = (null)
  prev2 = (null)
  prev' = (null)
  v = V0
Relations:
  next = {}
  next1 = {}
  val = {L0 -> V0}
  next' = {}
  next1' = {}
  val' = {L0 -> V0}
Skolem constants:
  c = L0
```

Figure 2: Counterexample to specification (2)

that, for most of the formulas we have analyzed (until now mostly drawn from abstract designs rather than code) small scopes often suffice.

Within a small scope, there are still many cases to consider. In a scope of 5, the *delete* procedure has about 2 billion possible inputs (without any constraints on the heap structure, and ignoring symmetry). Nevertheless, in this scope, and for 5 loop unrollings, our analysis finds the counterexamples in 4 seconds. With the additional preconditions added, so that no counterexamples are found, the resulting search of the entire space takes just over 2 minutes.

3 METHOD

3.1 Overview

Consider a path through the code of a procedure, consisting of a sequence of statements. At each point in the path – namely at the start, the end, and between each statement – we represent the execution state with a collection of set and relation variables. We then encode each statement as a constraint that relates the states before and after execution of the statement. The conjunction of these constraints characterizes, implicitly, the sequences of states that may hold along that path. Adding constraints that represent the negation of a specification now gives us a formula whose satisfying assignments are executions that violate the specification.

If we consider only at most some fixed number of loop iterations, any procedure has a finite number of paths, and we can create, by disjunction, a formula that represents all such paths. A naïve enumeration of paths, however, fails to exploit the large degree of sharing amongst paths, and suffers an exponential blowup in the number of branches. Our method therefore uses the control flow graph to construct the formula, and indeed most of our paper is concerned with this construction.

Since the formula language is undecidable, we bound the number of heap cells considered. This allows the formula in terms of sets and relations to be translated into a boolean formula that can be solved efficiently by an off-the-shelf SAT solver [24]. In effect, we are modelling the value of the heap as a collection of bit matrices, one for each field of a class. Suppose a class A has a field f of class B. Then the matrix for f has a 1 in the (i,j) position when the f field of the ith object of class A points to the jth object of class B.

We now explain the steps of the method in detail: how program state is modeled relationally; how the formula is constructed; and how it is solved. First, however, we give a brief outline of our specification language.

3.2 Specification Language

Alloy [22] is a language for writing formulas about sets and relations. Unlike other formal specification languages, Alloy is strictly first order, making automatic analysis possible [23, 24]. Alloy's syntax uses only ASCII characters, so it can be used conveniently for annotating code.

A syntax for the subset of Alloy we shall need here is given in Figure 3. (We have actually taken a few liberties to simplify the presentation. Alloy does not currently have boolean variables, nor the empty set constant; these deficiencies are worked around in our current implementation of the method by using a slightly less natural encoding.)

An Alloy formula is accompanied by declarations of free variables: sets, relations and booleans. Those sets that are not declared as subsets of other sets are called 'domains', and are uninterpreted. We will use one domain for each Java class: *List*, for example, will model the set of objects of class *List*.

The set declaration

s : t m

makes s a set of elements drawn from the set t, where t is either a domain or a set previously declared, and m is an optional multiplicity symbol. The multiplicity symbols (taken from regular expression syntax) are * (zero or more), + (one or more), ? (zero or one) and ! (exactly one). Omission is equivalent to *. So

p : List ?

declares *p* to be a set of lists containing zero or one element—that is, an 'option', which we will use to represent a possibly null list variable. The relation declaration

r : s m → t n

makes *r* a relation consisting of pairs whose first and second elements are drawn from the sets *s* and *t* respectively. The multiplicity symbols *m* and *n* are interpreted as in object model diagrams: there are *n* elements of *t* associated with each element of *s*, and *m* elements of *s* map to each element of *t*. So

next : List → List ?

declares next to be a partial function from lists to lists; it will be used to model a possibly null field next of a list whose value is a list.The syntax for formulas is largely conventional. In addition to the standard universal (*all*) and existential (*some*) quantifiers, there are quantifiers to indicate that there are no values satisfying the formula (*no*), exactly one value (*one*), and at most one value (sole).

The elementary formula

e1 in e2

is true when the expression *e1* denotes a subset of the set denoted by the expression *e2*. When *e1* is a singleton set, it can be viewed as a scalar and the formula as a membership test. The advantages of this scheme (making navigation expressions uniform and side-stepping the partial function problem) are explained in detail in [22].

All expressions denote sets: *e1 + e2* denotes the union of *e1* and *e2*, *e1 - e2* the difference, *e1* & *e2* the intersection. The symbol {} denotes the empty set. The *navigation expression* *e.r* denotes the image of the set *e* under the relation *r*. The relation *r* in this expression may more generally be a qualifier, with *~q* denoting the transpose of the relation denoted by *q*, *+q* the transitive closure, and **q* the reflexive-transitive closure.

3.3 Modelling State

For the purposes of exposition, we will use as our programming language the subset of Java shown in Figure 4. Note that there are no vectors or hash tables, although our method can handle them easily. It would also be straightforward to represent the heap model of C (without pointer arithmetic). Handling arrays will be possible so long as indices are not computed using complex arithmetic expressions. Constructors are easy to handle and omitted for lack of space; inheritance, however, is a problem we have not yet addressed.

The heap is modeled as a domain *C* for each class *C*, and a partial function for each field in a class. So, for example, if a class *A* has a field *f* of class *B*, we declare

f : A → B ?

problem ::= decl formula*
decl ::= set-decl | rel-decl | bool-decl
set-decl ::= set-var **:** *set-var mult*
rel-decl ::= rel-var **:** *set-var mult → set-var mult*
bool-decl ::= bool-var : **bool**
mult ::= **+** | **?** | **!**
formula ::= elem-formula | quant var [**:** *expr] '|' formula*
| formula bool-op formula | **¬** *formula | bool-var*
bool-op ::= **∧** | **∨** | **⇒**
quant ::= **all** | **some** | **no** | **one** | **sole**
elem-formula ::= expr **in** *expr | expr = expr | expr* **!=** *expr*
expr ::= set-var| set-var set-op set-var | expr **.** *qualifier |* **{}**
set-op ::= **+** | **-** | **&**
qualifier ::= rel-var | **'*'** *qualifier |* **+** *qualifier |* **~** *qualifier*

Figure 3: Syntax for a subset of Alloy

program ::= classdecl procdecl**
classdecl ::= **class** *class* **{** *class field* **,** **}**
procdecl ::= class **static** *proc* **(** *class var* **,** **)** **{** *stmt* **}**
stmt ::= var = expr | expr **.** *field* **=** *expr*
| **return** *expr*
| **while** *pred { stmt }*
| **if** *pred stmt stmt*
| stmt **;** *stmt*
expr ::= **null** *| var | expr* **.** *field*
pred ::= (expr **==** *expr) |* **!** *pred | pred* **&&** *pred*

Figure 4: Syntax for a subset of Java

so that for an object *a* of class *A*, *f* either maps *a* to some object *b* of class *B*, or does not map *a* at all (to model the case in which the *f* field of the object *a* is null). The environment is modelled by a collection of variables, one for each local variable or procedure argument. So, for example, for a variable *v* of type *C*, we declare

v : C ?

indicating that the value of *v* is a set of objects of class *C*, containing one or zero elements, admitting the possibility that *v* is null. The declarations for the delete procedure of Figure 1 will therefore include:

l : List ?
v : Val ?
next : List → List ?
val : List → Val ?

This scheme is just one way to model program state. Variables might instead be represented as atoms in their own right, and the stack as a function from variables to values.

Null might be represented as a special value. The choice affects the cost of the analysis, and the ease with which it may be instrumented (Section 3.10).

At each point in the execution of the procedure, there is a value for each variable and heap relation. We therefore generate constraint variables $v[i]$ and $f[i]$ for a program variable v and field f at each point i, although, as we shall see, it is often possible in the absence of modifications to share variables across program points.

3.4 Constructing the Computation Graph

Procedures are translated to formulas in two stages. First, an annotated control flow graph called a *computation graph* is constructed from the code. Second, the relational formula is extracted from the graph.

A computation graph is essentially a unrolled control flow graph. A node represents a program point, and an edge represents either a predicate test or an elementary program statement. There is a single entry node and a single exit node, every node sits on a path between entry and exit, and there are no cycles. Return statements and exceptional terminations are modelled by edges to the exit node.

A path through the graph from entry to exit will represent an execution of the procedure. Each node in the path represents control passing through a given textual point in the code; each predicate edge traversed represents an execution of that predicate in which it evaluates to *true*; and each statement edge traversed represents an execution of that statement. Not all paths through the graph are feasible, since the nodes themselves represent only the control portion of the state.

For a procedure without loops, the graph is the control flow graph of the program. If there is a loop, it is unrolled in the obvious manner. One unrolling, for example, applied to

a ; while (*p*) *s ; b*

results in the graph that would be obtained from the program

a ; if (*p*) *s ; assert !p ; b*

whose executions are required to have *p* be false at the assert.

The computation graph for one unrolling of the *delete* procedure of Figure 1 is shown in Figure 5.

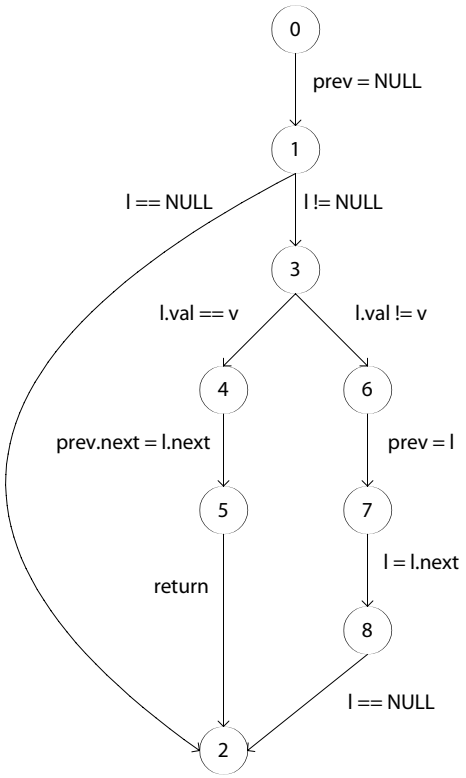

Figure 5: Computation graph for one unrolling of *delete*

3.5 Generating Variable Names

For each state component, we construct a labelling of the computation graph. The label of a node will be used to generate, at each node, a variable that represents the value of the state component there. (This scheme is similar to single static assignment [5], although we use disjunction to represent branches instead of phi functions, and introduce explicit frame conditions.)

The simplest approach is to generate a variable for each state component at each node. In this case, the labelling is the same for all components. Whenever a state component is unchanged between two adjacent nodes, we add an explicit equality between the variables at the two nodes. This results in far more variables than necessary, however.

We improve on this in two ways. First, we allow ourselves to assign the same label to nodes connected by a *non-modification edge* – an edge representing a predicate or statement that never changes the value of the state component. Second, we reuse labels across paths. Since no execution involves more than one path, we can use the same label twice so long as there is no path that passes through the two nodes.

For each state component, we assign labels to satisfy the following rules:

(a) on a given path, labels are distinct, although two nodes may have the same label if the subpath connecting them consists only of non-modification edges;

(b) for the graph as a whole, as few as possible labels are used.

A simple algorithm suffices: we label each node with an integer representing the length of the longest path to it from the entry node, treating non-modification edges as having length 0 and other edges as having length 1.

The result of this phase (shown for *delete* in Figure 6) is a mapping at each node from state components to constraint variables. For a state component c at node i, $c[i]$ is a constraint variable obtained by appending c's label at i to c's name.

The ovals in the Figure 6 show these variable names at each node. There are 3 variables for *prev*, corresponding to

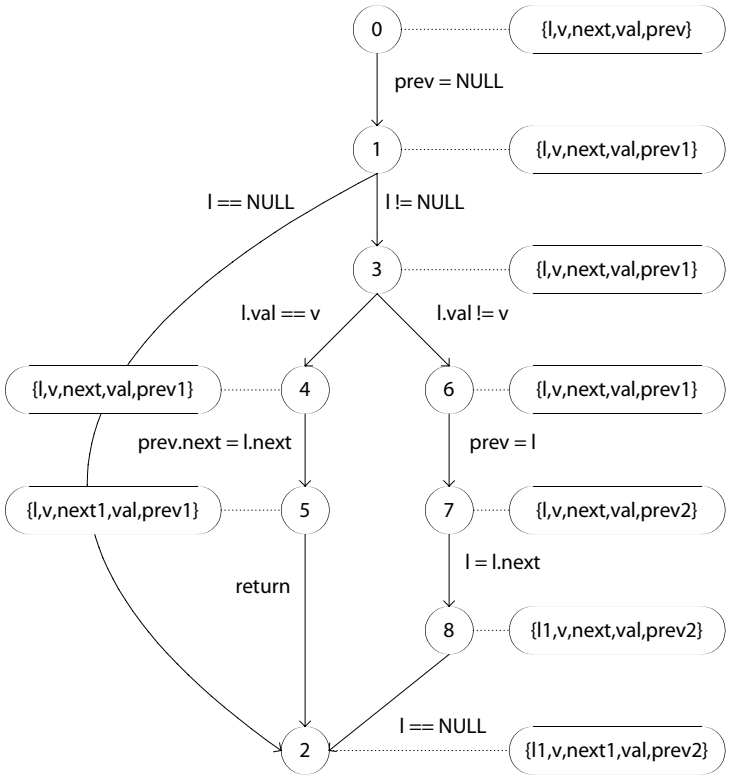

Figure 6: Generated variables for one unrolling of *delete*

S : *JavaStatement, Node, Node* → *AlloyFormula*
P : *JavaPredicate, Node, Node* → *AlloyFormula*
E : *JavaExpr, Node* → *AlloyExpr*

$S[\![v = e]\!]\ ij = v[j] = E[\![e]\!]\ i$
$S[\![e.f = t]\!]\ ij =$
$\quad (E[\![e]\!]\ i).f[j] = E[\![t]\!]\ i$
$\quad\quad \wedge (all\ o: Obj - (E[\![e]\!]\ i) \mid o.f[i] = o.f[j])$
$S[\![return\ e]\!]\ ij = result = E[\![e]\!]\ i$

$P[\![e1 == e2]\!]\ ij = E[\![e1]\!]\ i = E[\![e2]\!]\ i$
$P[\![!p]\!]\ ij = \neg P[\![p]\!]\ ij$
$P[\![p \&\& q]\!]\ ij = P[\![p]\!]\ ij \wedge P[\![q]\!]\ ij$

$E[\![v]\!]\ i = v[i]$
$E[\![NULL]\!]\ i = \{\}$
$E[\![e.f]\!]\ i = (E[\![e]\!]\ i).f[i]$

Figure 7: Translation functions

the initial value and the values following the two assignments to it, but only 2 variables for *next*, because it is modified by only a single statement. The components *val* and *v* are never changed, so only one variable is generated for each.

In order to allow the specification to refer to the state before and after execution without knowledge of the computation graph structure, a field component *f* is given the name *f'* on exit from the procedure. An extra variable is generated, along with a frame condition equating it to the variable name for *f* in the exit node. For a component that models a program variable, no primed constraint variable is created, since (under call-by-value) the value of a procedure argument is unmodified.

To avoid frame conditions, it helps to minimize the number of non-modification edges that connect nodes with distinct labels. The variable *prev2*, for example, is in fact not required, since *prev'* could take its place in nodes 2, 7 and 8. One could propagate labels across non-modification edges to do this, but our current implementation does not do so.

3.6 Encoding Control Flow

The structure of the computation graph is encoded as a boolean formula. Each edge e is given a boolean variable $trav[e]$; this variable will be true for executions in which the edge is traversed. For each node n with incoming edges $in(n)$ and outgoing edges $out(n)$, we generate the formula

$$\vee\{\ trav[e] \mid e \in in(n)\} \Rightarrow \vee\{\ trav[e] \mid e \in out(n)\}$$

which says that if an incoming edge is traversed, then some outgoing edge is traversed also.

Encoding the graph of Figure 5 produces:

$E01 \Rightarrow E12 \vee E13$
$E13 \Rightarrow E34 \vee E36$
$E34 \Rightarrow E45$
$E45 \Rightarrow E52$
$E36 \Rightarrow E67$
$E67 \Rightarrow E78$
$E78 \Rightarrow E82$

where $trav[e]$ is E with the labels of e's endnodes appended.

3.7 Encoding Data Flow

A formula is obtained from each edge of the computation graph that encodes its dataflow. The encoding is specified with three translation functions (Figure 7) that act on the label of the edge, and are parameterized by its start and end nodes. For a statement edge, the function S is applied; for a branching edge labelled with a predicate, the function P is applied; E is an auxiliary function used to translate expressions in both predicates and statements.

The first rule for S, for example, says that an edge labelled with the statement $v = e$ for Java variable v and expression e, and which connects node i to node j, is translated into a formula that equates the Alloy variable that represents v at j with the Alloy expression that denotes the value of the expression e at node i. The second rule, for the field-setter $e.f = t$, equates the value of the expression t in i with the value of the field f in j of the object denoted by e in i. It also adds a frame condition saying that the f field of no other object changes.

The formula obtained from an edge label is only relevant if that edge is executed. So for an edge e whose label is translated to formula F, the final formula is

$$trav[e] \Rightarrow F$$

which says that the dataflow encoded in F applies when the edge is traversed.

For example, the statement

$$l = l.next$$

on the edge between nodes 7 and 8 gives

$$E78 \implies l1 = l.next$$

3.8 Frame Conditions

When an edge connects nodes that assign a different label to a state component, but that component is not modified by the statement associated with the edge, a frame condition is generated. If the component is a Java variable v, and the edge connects node i to node j, the condition is

$$v[i] = v[j]$$

If the component is a field f, the condition is

$$all\ o \mid o.f\ [i] = o.f\ [j]$$

The set of components not modified by a statement is easily obtained: for $v = e$, it includes all components bar v; for $e.f = t$, it includes all components bar f.

For example, nodes 8 and 2 are connected by a non-modification edge for *next*, so we would generate the frame condition

$$E82 \Rightarrow all\ o \mid o.next = o.next1$$

3.9 Formula Extraction and Analysis

Putting things together, the *code formula* representing the entire graph is obtained by conjoining the control flow constraints, the dataflow constraints and the frame conditions. The result for one unrolling of *delete* is shown in Figure 8. Given a specification with a precondition *Pre* and a postcondition *Post* (which may mention both pre- and post-states), the *specification formula* is

$$Pre \Rightarrow Post$$

$$
\begin{array}{l}
E01 \\
E01 \Rightarrow prev1 = \{\} \\
E01 \Rightarrow E12 \vee E13 \\
E12 \Rightarrow l = \{\} \wedge prev2 = prev1 \wedge l1 = l \wedge all\ e \mid e.next1 = e.next \\
E13 \Rightarrow l\ != \{\} \\
E13 \Rightarrow E34 \vee E36 \\
E34 \Rightarrow l.val = v \\
E36 \Rightarrow l.val\ != v \\
E34 \Rightarrow E45 \\
E45 \Rightarrow prev1.next1 = l.next\ \wedge\ all\ e{:}\ List - prev1 \mid e.next1 = e.next \\
E45 \Rightarrow E52 \\
E52 \Rightarrow prev2 = prev1 \wedge l1 = l \\
E36 \Rightarrow E67 \\
E67 \Rightarrow prev2 = l \\
E67 \Rightarrow E78 \\
E78 \Rightarrow\ l1 = l.next \\
E78 \Rightarrow E82 \\
E82 \Rightarrow l1 = \{\} \wedge all\ e \mid e.next1 = e.next \\
prev' = prev2 \\
all\ e \mid e.next' = e.next1 \\
all\ e \mid e.val' = e.val
\end{array}
$$

Figure 8: Formula for code of *delete*, one unrolling

whose negation is conjoined to the code formula C, giving

$$Pre\ \wedge Code \wedge \neg Post$$

Models of this formula represent executions of the code in which the precondition holds but the postcondition does not: in other words, cases for which the code is supposed to behave correctly, but fails to.

The formula is solved using our constraint analyzer for Alloy formulas [23, 24]. The only additional input beyond the formula and the declarations of the state components is the choice of *scope*, bounding the number of elements in each type – in this case the representing the set of objects in each class.

If the analysis finds a model, a counterexample trace is easily reconstructed from it. We examine the values of the boolean variables $trav\ [e]$ to extract the path through the graph; the value of each state component at each node is then found by examining the variable with the appropriate label.

If the analysis fails to find a model, no error has been found within the given scope. One can increase the scope, in the hope of finding an error that requires a larger heap. Soon, however, the analysis becomes intractable. We have yet to determine how large a scope is required to find most errors; for the benchmark procedures, a scope of 1 sufficed to find all known bugs.

3.10 Null Dereferences

If the procedure being analyzed has executions that dereference null, the analysis described so far may miss errors (but it will not generate spurious reports). To handle potential null

dereferences, we instrument the code with an extra state component, *NullDerefs*, whose value at any program point holds the set of objects with a null field that has been improperly dereferenced.

Let *Refs* $[\![e]\!]$ be the set of expressions denoting objects whose fields are dereferenced when e is evaluated. If e is $v.f.g.h$, for example, *Refs* $[\![e]\!]$ would include v, $v.f$ and $v.f.g$. Whenever an expression $e.f$ appears in a statement labelling an edge from i to j, we generate the formula

$$NullDerefs[j] = NullDerefs[i] + \{\, o \mid o = Refs\,[\![e]\!] \wedge E[\![e]\!] = \{\}\}$$

which adds to the null dereference set each object o which has a null field that results in e evaluating to null. To check the procedure, we generate a specification saying that the final value of *NullDerefs* is empty. A counterexample will show not only at which point in the program the null dereference occurred, but also which object had the null field. (This scheme does not catch dereferencing of null-valued variables. To track variables, one could model them explicitly, and represent the stack as a function from variables to objects.)

A subtle problem arises when a procedure is faulty. If the expression e evaluates to null in Java, then in the formula obtained by translating

$$e.f = x$$

the expression corresponding to e will denote the empty set, and its image under the field relation will likewise be the empty set. Unless the right-hand expression x also evaluates to null, the formula as a whole will be false, and the path involving this case will not be considered!

To prevent this in a clean and systematic fashion, we could add an edge from the statement to the exit node of the procedure, faithfully modelling a runtime exception when the null dereference occurs. But this complicates the construction, and in practice, a simple trick works well. We translate the right-hand side expression to

$$x \;\&\; \{o : Obj \mid e \mathrel{!}= \{\}\}$$

The expression $\{o : Obj \mid e \mathrel{!}= \{\}\}$ denotes the set of all objects when e is non-empty, and the empty set otherwise. Intersecting it with x therefore gives x, as before, when e is not a null dereference, and gives the empty set, making the formula vacuously true, when it is.

Similar instrumentation can be added to account for other properties that involve abnormal terminations. Memory leaks, for example, could be handled by tracking which objects are free; and erroneous typecasts could be caught by testing membership of objects in the sets representing the expected classes.

4 EXPERIMENTS

We applied our method to a suite of 8 procedures taken from [9], which include sample programs taken from [12] and [25]. These all involve destructive updates of linked lists, and vary from 7 to 30 lines in length.

The formulas were obtained mechanically from the code using a prototype translator. Translation time is insignificant. Because Alloy has no boolean type and is not designed to handle large numbers of disjunctions, we were forced to represent the *trav* variables and their formulas in a rather unnatural (and perhaps less efficient) manner than described above. This problem is not hard to fix.

All analyses were performed on a 233MHz Pentium 2 with 132MB of memory.

We wrote assertions to check two kinds of property. First, we analyzed the procedures for the anomalies considered by [9]: namely null dereferences and creation of cyclic structures. (Since we translated the sample programs from C to Java, we did not consider the analysis of memory leaks, even though it is easily handled by our method).

Second, we specified the obvious user-defined properties for each procedure. These included simple set inclusion properties: for example, that *reverse* and *rotate* do not change which cells are in the list; that *merge* yields a list with the union of the cells in each input list; and that *delete* removes the given cell. We also checked more complex properties: for example that *reverse* actually reverses the list and that *merge* produces a sorted output list from two sorted input lists.

Our analysis detected all the anomalies previously detected, as well as an anomaly not mentioned in the previous papers: that *merge* creates a cyclic list if its arguments are aliased. All the anomalies were found in a scope of 1. In a scope of 3 – that is, 3 list objects and 3 value objects – finding a counterexample takes no more than a second for any of the examples.

When there is no counterexample, and for a scope of 3, the analysis takes less than a second in all cases except for one: checking that the result of the *merge* procedure is a list whose elements include the elements of the argument lists p and q:

$$result.^*next = p.^*next + q.^*next$$

The timings for this analysis for a variety of scopes are shown in Table 1. The column marked *#iters* gives the number of loop iterations considered: loops were unrolled to match the scope, since otherwise the longest possible argument list cannot be traversed. The column marked *#bits* gives the number of boolean variables used to encode the set and relation variables that represent the values of state components; the boolean formula presented to the solver typically has many more variables because of the introduction of variables in the conversion to conjunctive normal form. The last column gives the time taken to exhaust the search.

Table 1: Performance for hardest problem

scope	iters	bits	time
1	1	103	0s
2	2	133	2s
3	3	163	12s
4	4	193	6m 23s

5 RELATED WORK

5.1 Testing

Our method might be viewed as a form of testing, but it differs from testing in several important respects:

- It can be applied to code in which calls are made to procedures for which specifications but no code is available. This means that programs can be analyzed during construction before they are complete. Also, programs that make use of an API can be fully analyzed even though the actual code of the API is unavailable, and libraries that are implemented at a lower level (eg, as native code) can be accounted for.
- It generally considers many more cases than are feasible in conventional testing. For the benchmark examples, our method can consider millions of possible inputs to a list processing procedure in only a couple of seconds.
- Our method does not execute the code, but rather propagates values in a goal-directed fashion. For example, if the specification requires that the result of the procedure be an acyclic list, our method might effectively start by assuming a cyclic list as the result, and search backwards to an appropriate input from there. In fact, as we shall explain, the search is performed on a boolean formula and so predicting exactly how it proceeds is hard, since it depends on the heuristics of the solver and the exact structure of the formula. Nevertheless, this intuition helps to explain why our method can often find elaborate counterexamples with relatively little search.
- Any property of the program state that can be expressed in our specification language can be analyzed. For example, our method could be used to detect memory leaks in C programs, simply by giving distinguishing freed and unfreed cells, and checking at the end of the procedure that every unreachable cell has been freed. Although testing can analyze such properties, the instrumentation needed is often elaborate.

Our method is not intended to replace testing. Our specification language is designed for object model properties: namely what objects exist, how they are classified grossly into sets, and how they are related to one another. It is not designed for arithmetic properties, so our method is not suitable for analyzing numerical algorithms. We do not address concurrency, although we should be able to analyze properties that are important in a concurrent setting, such as ownership of locks and containment of objects. Moreover, like a static analysis, our method treats program statements as if they have their advertised semantics, so unlike testing, will not detect errors due to faulty compilation or to unexpected interactions with the operating system, device drivers, etc.

5.2 Shape Analysis

Classic shape analyses (eg, [5,14]) simulate the execution of the program text over an abstract state. Each instance of the state represents a (usually infinite) set of concrete states. By ensuring that the abstraction is *conservative*—that is, the set of concrete states always includes the states that might arise in an actual execution—the analysis can effectively prove properties for all executions. Our method differs in the following respects:

- As discussed above, our method answers "no" or "maybe", rather than "yes" and "maybe". Failure to detect a bug does not mean that the program is correct, but error reports are not spurious.
- Shape analyses are typically designed to analyze a program for a small, fixed repertoire of properties. Our specification language allows the user to express arbitrary structural property of the state. Furthermore, shape analyses do not usually relate states at different points; they indicate what properties the state has at a given point in the code, but do not, for example, infer properties about how the initial and final states are related.
- Most shape analyses are based on a form of abstract interpretation in which the shape of the heap at a given point is obtained by applying a transfer function to the shapes of the heaps at immediately preceding points. As a consequence, such analyses cannot handle procedure calls by using their specifications instead, unless the specifications are given as abstract programs. Moreover, since the analysis cannot be reversed, it is not possible to derive a counterexample trace when a property is found not to hold.
- The abstractions used in shape analyses lose a lot of information. In short, the cost of soundness is a large number of false negatives. In a compiler, these simply result in missed opportunities for optimization, but in a software engineering tool, they cause spurious error reports. Whether an abstraction succeeds is usually highly dependent not only on the behaviour of the code, but also how it is written; a tool based on shape analysis is thus likely to be unpredictable. There are no conservative analyses, to our knowledge, that can analyze the full range of properties our specification language can express without often producing spurious reports.
- Shape analyses are tailored to particular properties. If a new property is to be analyzed, the abstraction must usually be altered. In our method, the transformations performed on the code itself are independent of the specifica-

tion against which it is being checked, and the incorporation of the specification itself is by a simple logical operation, with no change to the representation of the state.

Recently, Sagiv, Reps and Wilhelm [29] have developed a *parametric shape analysis* (PSA). It does not fundamentally overcome any of these problems: it cannot incorporate specifications in the code, for example, or generate concrete counterexamples, and the analysis must be tailored (using 'instrumentation predicates') to the property being checked. Nevertheless, the underlying mechanism can accommodate a wider range of properties than traditional shape analyses—in principle the same as ours, since their property language is a first-order logic with transitive closure. Also, unlike other shape analyses, PSA can sometimes give a sound 'no' answer to a property check.

5.3 Lightweight Static Analyses

A number of analyses based on augmented type systems have been developed that are less powerful (but more tractable) than shape analyses, such as refinement types [13] and soft types [4]. None of these are powerful enough to analyze the kinds of properties we are concerned with here, since they do not correlate the values of different variables. Many other analyses, such as set constraints [16], comments analysis [17], LCLint [12], Inscape [28], and Aspect [18] are limited for the same reason.

5.4 Symbolic Execution & Theorem Proving

A decision procedure for pointer-properties of loop-free programs has been developed by Jensen et al [25]. A Hoare triple yields a proof obligation that is translated into a second-order monadic logic, with a potentially non-elementary blowup. In practice, the worst case blowup is not observed, and the same procedures we have analyzed here are handled in times ranging from 25 to 94 seconds. The analysis can prove properties, and can generate concrete counterexamples. The user supplies loop invariants. It is not clear whether the method can be extended to heap structures that are not linear or tree-like.

The Extended Static Checker (ESC) [8] uses a powerful, tailored theorem prover to check code against user-supplied specifications. It has been applied mainly to showing the absence of flaws such as out-of-bounds array access, null pointer dereferencing and unsound use of locks. It has not been used to check the kind of heap structuring properties our method addresses. Error reports may be spurious, although this appears to be rare in practice. More remarkably, omission of error reports does not imply correctness. The conventional rules about modifies clauses and beneficent side effects turn out to be unsound [27]. ESC's developers found that correcting them increases the number of spurious reports unacceptably.

The PREfix tool, developed by Jon Pincus of Microsoft Research, detects anomalies by a symbolic execution of the code [26]. Carefully judged heuristics allow it to detect many errors without generating too many spurious reports; it has apparently found thousands of anomalies in the code of Windows 95.

5.4 Model Checking

Currently, there is much interest in applying model checking technology to code. The Bandera project (at Kansas State, Hawaii and UMass) is developing a toolkit that extracts finite state machines from code using program slicing and shape analysis [6], and then applies off-the-shelf model checkers. The SLAM project (at Microsoft Research) has developed a strategy that combines symbolic execution and model checking that allows the extracted model to be refined incrementally. The model checking algorithm itself is novel, and uses context-free language reachability to handle recursive procedure calls [2]. The FLAVERS project (at UMass, Amherst) uses dataflow analysis to analyze extracted state machines, and also supports incremental refinement [10].

All of these have been concerned primarily with the analysis of event sequences rather than data structures. Where analysis of data structures has been used, its purpose has been to simplify the state machine to be analyzed, rather than to provide feedback to the user about the data structures themselves. The Java Pathfinder project (at NASA Ames) also focuses on the checking of event properties, but, more in common with our method, involves data structures in the model checking analysis [15]. A Java program is essentially simulated for all possible interleavings of threads, using dSPIN, a variant of the SPIN model checker that offers dynamic allocation of heap cells.

Pathfinder is designed for checking concurrency properties. How well it would work for data structure properties is unclear. Being simulation-based, Pathfinder is a whole-program analysis: it cannot analyze a procedure in isolation, or use implicit specifications for called procedures.

SAT solvers have been used to find executions of a system in planning [11], in checking software specifications [20,21], and more recently in linear temporal logic checking [3]. The idea of representing a sequence of operations as a single formula is much older, and is present in the Z specification language [30] for example. The use of a constraint solver to find a counterexample trace from a fixed sequence of operations is illustrated in [19]. Considering a few unwindings of a loop is an idea familiar from program testing; the idea of bounding execution paths is found in the earliest papers on symbolic execution [26].

6 CONCLUSIONS

We have described a simple but expressive specification notation in which data structures are treated as relations (or equivalently graphs), and a fully automatic analysis that can expose non-trivial errors and give concrete counterexamples.

The method involves two steps: translation of the code into a relational formula, and an attempt to find models of the relational formula. The efficiency of the analysis, and its ability to consider huge numbers of cases, comes from the power of modern SAT solvers, which lie at the core of our analysis tool.

The analysis takes two parameters: the number of loop unrollings, which affects the translation into a relational formula, and the scope (ie, the number of heap cells), which affects the translation of the relational formula into a boolean formula.

We have shown that our method is at least able to detect all the anomalies in a benchmark suite of programs previously found by shape analyses, but that in addition we are able to check user-defined properties. Since our analysis treats even elementary program statements as if they were declarative specifications, we are hopeful that it will extend to the analysis of code in terms of abstract sets and relations specified in an API.

The effectiveness of our method lies in the *small scope hypothesis*: that even though, for any given scope k, one can construct a program with a bug whose detection requires a scope of k+1, in practice, many bugs will be detectable in small scopes. Indeed, all of the anomalies previously identified in the benchmark suite can be demonstrated with counterexamples that use a single list cell.

To ensure that small scopes will suffice, we will need to overcome at least two obstacles. First, there may be resource boundaries: this is why programs often fail when a structure's size grows just beyond a power of 2. We will need to treat any such bounds as parameters of the analysis, so that resource overflow can be simulated in small scopes. Second, a larger scope may be required simply to populate a collection of irrelevant structures. Some kind of slicing may address this problem.

It will not be practical when analyzing large programs to model all datatypes. In practice therefore, there will be information in the state that is abstracted away in the analysis, and a counterexample may be generated for an infeasible path. It remains to be seen whether this will be a problem in practice.

ACKNOWLEDGMENTS

This work benefited greatly from discussions with Tom Ball, Nurit Dor, Fritz Henglein, Rustan Leino, Tom Reps, Martin Rinard and Mooly Sagiv; from David Karger's help in developing the labelling algorithm; from the contributions of Ian Schechter and Ilya Shlyakhter to the Alloy Constraint Analyzer upon which this work is based; and from the helpful comments of Sarfraz Khurshid and the anonymous reviewers on an earlier draft of the paper. The research was supported in part by the MIT Center for Innovation in Product Development, funded under NSF Cooperative Agreement Number EEC-9529140, and by a grant from the Nippon Telephone and Telegraph Corporation.

REFERENCES

[1] F.E. Allen. Control Flow Analysis. *Proceedings of a Symposium on Compiler Optimization*. SIGPLAN Notices 5, 7, July 1970, pp. 1–19.

[2] Thomas Ball and Sriram K. Rajamani. *Boolean Programs: A Model and Process for Software Analysis*. MSR Technical Report 2000-14, Microsoft Research, Redmond WA, February 2000.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. *Tools and Algorithms for the Analysis and Construction of Systems* (TACAS'99), LNCS 1579, Springer-Verlag, 1999.

[4] Robert Cartwright and Matthias Felleisen. Program Verification Through Soft Typing. *ACM Computing Surveys*, 28(2), pp. 349-35, 1996.

[5] D.R. Chase, M. Wegman and F. Zadeck. Analysis of Pointers and Structures. *Proc. Conf. on Programming Language Design and Implementation*, pp. 226–228, ACM Press, 1990.

[6] James C. Corbett. Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. *ACM Transactions on Software Engineering and Methodology*, 9(1), 2000, pp. 51–93.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, Vol. 13, October 1991, pp. 451–490.

[8] D. Detlefs, K. R. Leino, G. Nelson, and J. Saxe. *Extended static checking*. Technical Report 159, Compaq Systems Research Center, 1998.

[9] Nurit Dor, Michael Rodeh & Mooly Sagiv. Detecting Memory Errors via Static Pointer Analysis. *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (PASTE '98) Montreal, June 1998.

[10] Matthew Dwyer and Lori Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. *Proc. Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, December 1994, pp. 62–75.

[11] Michael D. Ernst, Todd D. Millstein and Daniel S. Weld. Automatic SAT-Compilation of Planning Problems. *Proc. 15th International Joint Conference on Artificial Intelligence* (IJCAI-97), Nagoya, Aichi, Japan, August 1997, pp. 1169–1176.

[12] David Evans. Static Detection of Dynamic Memory

Errors. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1996.

[13] Tim Freeman and Frank Pfenning. Refinement Types for ML. *Proc. ACM Symposium on Principles of Programming Languages*, ACM Press, 1991.

[14] R. Ghiya and L. Hendren. Putting Pointer Analysis to Work. *Proc. ACM Symposium on Principles of Programming Languages*, ACM Press, 1998.

[15] Klaus Havelund and Tom Pressburger. Model Checking Java Programs Using Java Pathfinder. *International Journal on Software Tools for Technology Transfer,* 2(4), Springer-Verlag, April 2000.

[16] Nevin Heintze. Set Constraints in Program Analysis. 1993. *Proc. 10th International Symposium on Logic Programming*, D. Miller (Ed.). MIT Press, 1993.

[17] William E. Howden. Comments Analysis and Programming Errors. *IEEE Transactions on Software Engineering*, SE-16/1, January 1990.

[18] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 2, April 1995, pp. 109-145.

[19] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 484–495.

[20] Daniel Jackson. *Boolean Compilation of Relational Specifications.* Technical Report MIT-LCS-TR-735, MIT Lab for Computer Science, Cambridge, MA, December 1997.

[21] Daniel Jackson. An Intermediate Design Language and its Analysis. *Proc. ACM SIGSOFT Foundations of Software Engineering*, Orlando, Florida, 1998.

[22] Daniel Jackson. *Alloy: A Lightweight Object Modelling Notation*. Technical Report 797, MIT Laboratory for Computer Science, Cambridge, Mass, February 2000. Available at: http://sdg.lcs.mit.edu/~dnj/abstracts.html#alloy.

[23] Daniel Jackson, Ian Schechter and Ilya Shlyakhter. Alcoa: the Alloy Constraint Analyzer. *Proc. International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[24] Daniel Jackson. Automating First-Order Relational Logic. To appear, *Proc. ACM SIGSOFT Foundations of Software Engineering*, San Diego, CA, November 2000.

[25] Jakob L. Jensen, Michael E. Jorgensen, Nils Klarlund and Michael I. Schwartzbach. Automatic Verification of Pointer Programs using Monadic Second-Order Logic. *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, 1997.

[26] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, vol. 19, no. 7, July 1976, pp. 385-394.

[27] K. Rustan M. Leino. *A myth in the modular specification of programs.* KRML 63 –0, Compaq Systems Research Center, Palo Alto, CA, November 1995.

[28] Dewayne Perry. The Logic of Propagation in the Inscape Environment. *Proc. 3rd ACM Symposium on Software Testing, Analysis and Verification* (TAV3), Key West, Florida, Dec. 1989.

[29] Mooly Sagiv, Tom Reps and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *Proc. ACM Symposium on Principles of Programming Languages*, San Antonio, TX, Jan. 20-22, 1999, ACM, New York, NY, 1999.

[30] J. Michael Spivey. *The Z Notation: A Reference Manual*. Second ed, Prentice Hall, 1992.