

# Checking Properties of Heap-Manipulating Procedures with a Constraint Solver

Mandana Vaziri & Daniel Jackson

Laboratory for Computer Science  
Massachusetts Institute of Technology  
Cambridge, Massachusetts, USA  
{vaziri,dnj}@lcs.mit.edu

**Abstract.** A method for finding bugs in object-oriented code is presented. It is capable of checking complex user-defined structural properties – that is, of the configuration of objects on the heap – and generates counterexample traces with no false alarms. It requires no annotation beyond the specification to be checked, and is fully automatic.

The method relies on a three-step translation: from code to a formula in a first-order relational logic, then to a propositional formula, and finally to conjunctive normal form. An off-the-shelf SAT solver is then used to find a solution that constitutes a counterexample.

This underlying scheme, presented previously, does not scale readily. In this paper, we show how a suite of optimizations results in much improved scalability. The optimizations are based on a special treatment of relations that are known to be functional, and target all steps. The effect of the optimizations is demonstrated by application to the analysis of a red-black tree implementation.

## 1 Introduction

An earlier paper [12] presented an analysis for finding bugs in object-oriented code using a constraint solver. The analysis takes as input a user-specified property, and a program fragment, and its output is either a counterexample – an execution through the code violating the property – or “inconclusive”. The user may specify properties relating variables in different states, and structural properties of the heap. The method requires no intermediate code annotations and is fully automatic. When a counterexample is found, it is guaranteed to be a feasible execution of the code.

The analysis gives a bounded interpretation to a fragment of Java code, where loops are unwound up to a certain number, and procedure calls are inlined, and translates it into Alloy [11], a modelling language based on first-order logic. User-defined properties are also expressed in Alloy, which allows succinct declarative expressions of complex structural properties. For example, the fact that a list is acyclic can be expressed using the transitive closure operator, and that red-black trees have the same number of black nodes on each path is expressible with set cardinalities. Given an Alloy model for the program, the Alloy Analyzer [10] is

then used, together with a user-provided bound on the number of heap cells, to check properties, and to find counterexamples.

Our method differs from other verification approaches in that it targets properties of the heap. It considers all the possible initial configurations within finite bounds. If there is a property violation, it will determine the initial configuration responsible for it, as part of the counterexample trace. The analysis typically accounts for billions of cases, which would not be feasible with testing alone. The SAT solver – the Alloy Analyzer’s core engine – works in a goal-oriented fashion. Since it tries to satisfy a boolean formula, it does not go through all the executions in turn, and may effectively search multiple executions at once. Our approach also does not perform approximations beyond considering a finite instance of the code. This means that there are no false error reports.

The scheme relies on a three-step translation: a code fragment is first translated into Alloy, a first-order relational logic (Section 2), then into propositional logic (Section 3), and finally into conjunctive normal form (CNF), which is then presented to a SAT solver. We can use the fact that a field is functional, meaning that it can point to at most one object, to optimize these steps (Section 4). Our optimizations are a suite of simple but judiciously chosen logical simplifications. They are not all incremental, meaning that implemented in isolation some do not result in an improvement. Together, however, they bring a marked improvement in scalability. We illustrate their effect by analyzing a red-black tree implementation (Section 5). The paper concludes with a discussion of related work (Section 6).

## 2 Encoding Object-Oriented Code in Alloy

In this section, we overview our basic translation of Java code into Alloy [12].

### 2.1 Illustration

Consider the `swapTail` procedure (Figure 1), which purportedly takes two linked lists and swaps their tails. We use our analysis to check whether the `swapTail` procedure preserves the property that its inputs are acyclic. We write the Alloy specification shown in Figure 1. The assertion states that: for all `Lists l` and `m`, if they are both acyclic in the initial state, and `swapTail(l,m)` is called, then `m` is acyclic in the post state, which is indicated by the prime sign on the last appearance of the function `Acyclic`.

The auxiliary function `Acyclic(x: List)` defines the constraint that a list is acyclic, by stating that for all `ListElems e` reachable from `x.first`, where `x.first` is included, `e` is not reachable from itself.

The command that follows instructs the tool to use 2 heap cells per type and 1 iteration. The analysis produces the counterexample in Figure 1, where only a part of it is shown. Black circles represent heap cells of type `List`, and white ones are of type `ListElem`. Arrows represent fields. In the pre-state, list `m` is a list of one element, and `l` of two, and they share an element. In the post-state, `m`

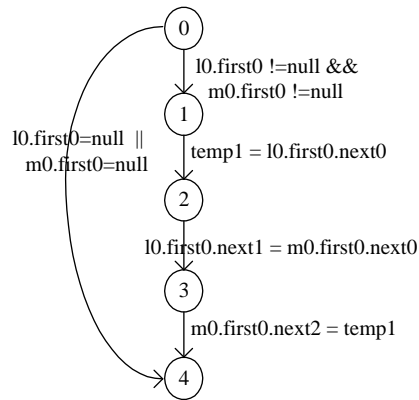
### Code

```

class ListElem {
  int val;
  ListElem next;
}
class List {
  ListElem first;
0 static void swapTail(List l, List m){
1   if (l.first != null
        && m.first != null) {
2     ListElem temp = l.first.next;
3     l.first.next = m.first.next;
4     m.first.next = temp;
   }
}

```

### Computation Graph



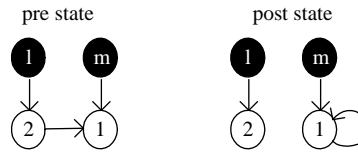
### Property

```

fun acyclic(x: List) {
  all e: x.first.*next |
    e !in e.^next
}
assert A {
  all l, m: List |
    acyclic(l) && acyclic(m)
    && swapTail(l,m) => acyclic'(m)
}
check A for 2 List, 2 ListElem, 1 iteration

```

### Counterexample



### Free Variables

$first_0 : List \rightarrow! ListElem$   
 $next_0, next_1, next_2 : ListElem \rightarrow! ListElem$   
 $l_0, m_0 : List$   
 $temp_0, temp_1 : ListElem$   
 $E_{01}, E_{12}, E_{23}, E_{34}, E_{04} : Bool$

### Control Flow

$E_{01} \ || \ E_{04}$   
 $E_{01} \ \Rightarrow \ E_{12}$   
 $E_{12} \ \Rightarrow \ E_{23}$   
 $E_{23} \ \Rightarrow \ E_{34}$

### Data Flow

$E_{01} \ \Rightarrow \ l_0.first_0 \neq null \ \&\& \ m_0.first_0 \neq null$   
 $E_{04} \ \Rightarrow \ l_0.first_0 = null \ || \ m_0.first_0 = null$   
 $E_{12} \ \Rightarrow \ temp_1 = l_0.first_0.next_0$   
 $E_{23} \ \Rightarrow \ l_0.first_0.next_1 = m_0.first_0.next_0 \ \&\&$   
 $all \ o:ListElem-l_0.first_0 \ | \ o.next_1 = o.next_0$   
 $E_{34} \ \Rightarrow \ m_0.first_0.next_2 = temp_1 \ \&\&$   
 $all \ o:ListElem-m_0.first_0 \ | \ o.next_2 = o.next_1$

### Additional Frame Condition

$E_{04} \ \Rightarrow \ next_2 = next_0 \ \&\& \ temp_1 = temp_0$

Fig. 1. An Example of a Procedure and its Analysis

is a list with an element whose next field points to itself, and is therefore cyclic, violating the assertion.

## 2.2 Extracting a Computation Graph

We translate a Java procedure into an Alloy formula whose models correspond to executions. It is composed of two subformulas, encoding the control and data flow of the procedure.

We start with the procedure’s control flow graph (CFG), with nodes representing its control points, and edges labeled with either Java statements or control predicates. We unroll the loops in the CFG up to a number of iterations (selected by the user), and inline procedure calls, to obtain a *computation graph*. For example, one unrolling of `a; while(p) s; b` gives the graph one would obtain as the standard CFG of `a; if(p) s; assert !p; b`.

We rename variable and field names, in such a way that no path has two assignments to the same variable or sets the same field name, but parallel paths may share names (just as in single static assignment [2], but without  $\phi$  functions). Renaming is done by providing an index for each variable at each node in the computation graph. In what follows we use  $v_i$  to denote the name of variable  $v$  at node  $i$ .

Figure 1 shows the computation graph corresponding to the `swapTail` procedure. The edge between nodes (0) and (4) is traversed when the condition of the if statement is false.

## 2.3 Encoding the State, Control, and Data Flow

Variables in the Alloy formula are used to encode the state at the various control points of the procedure. We model a field with a total function from the class to which it belongs to its declared type. Each type has a special atom representing null. Local variables and formal parameters are modelled as scalars, which are represented by singleton sets in Alloy.

In addition, the variables of the Alloy formula include a boolean variable<sup>1</sup>  $E_{ij}$  for each edge from node  $i$  to  $j$  in the computation graph. These are used to encode the control and data flow, and indicate whether an edge is traversed during an execution. The free variables for `swapTail` are shown in Figure 1, where a declaration of the form  $f : A \rightarrow !B$  says that  $f$  is a relation from  $A$  to  $B$ , restrained to be a total function from  $A$  to  $B$ , and  $a : A$  declares  $a$  to be a scalar of type  $A$ .

The control flow is encoded with a formula that captures when an edge is traversed. For each node  $i$ , let  $in(i)$  be the set of nodes having an outgoing edge to  $i$ , and  $out(i)$  the set of nodes having an incoming edge from  $i$ . For each node  $i$ , we produce  $\bigvee\{E_{ji} | j \in in(i)\} \Rightarrow \bigvee\{E_{ik} | k \in out(i)\}$ , and the formula encoding the control flow is the conjunction of these formulas. These mean that if some node’s incoming edge is traversed then some of its outgoing edges are also

<sup>1</sup> Alloy does not have built-in booleans, but these can be easily encoded with sets.

traversed. Infeasible paths are ruled out because some of the edges are labeled with control predicates, and these appear in the formula that encodes the data flow presented below. Note that if more than one outgoing edge is traversed, the constraint solver may generate an instance corresponding to more than one execution. But all these executions are feasible, so the analysis outputs all of them. In the case of `swapTail`, the formula encoding control flow is shown in Figure 1.

We encode the data flow for each edge with a formula that indicates how variables are related before and after the statement corresponding to that edge is executed. For each edge  $e$  from node  $i$  to  $j$ , we produce a formula:  $E_{ij} \Rightarrow t$ , where  $t$  is the translation of the Java statement corresponding to  $e$  into first-order logic. The formula that encodes the data flow is then the conjunction of these formulas. This means that whenever  $e$  is traversed, the effect of the Java statement encoded by  $t$  is observed. The translation rules for Java statements are given in our earlier paper [12]. In the case of `swapTail`, the formula encoding data flow is shown in Figure 1.

In Alloy, the expression  $a.r$ , where  $a$  is a set and  $r$  is a relation, denotes the relational image of  $a$  under  $r$ . So `l0.first0` denotes the image of `l0` under function `first0`. The formula `all o: ListElem - l0.first0 | o.next1 = o.next0` is a *frame condition*. It means that for all `ListElem`s other than `l0.first0`, the `next` relation remains the same. Alloy is a declarative language in which variables that are left unconstrained can take on any value. Frame conditions are then needed to say that certain variables remain the same when an update happens.

Finally, we conjoin a set of additional frame conditions with the formula encoding data flow. When an edge connects nodes  $i$  and  $j$  that assign a different index to a field or variable  $v$ , but  $v$  is not modified by the statement associated with the edge, we produce the frame condition:  $E_{ij} \Rightarrow v_j = v_i$ . In the case of `swapTail`, the additional frame conditions are shown in Figure 1, and say that whenever `E04` is traversed, the `next` relation, and the `temp` variable, remain the same.

### 3 Translation to Propositional Logic

In Alloy, every type consists of a set of atoms. The values of variables are relations, which are sets of tuples of atoms, and sets are treated as degenerate relations, consisting of a set of unary tuples. The user provides a bound  $n$  on the number of heap cells for each class to our analysis, and this is used to set *scopes* for the Alloy Analyzer, i.e. the number of atoms of each type. The analyzer uses the scope to translate an Alloy formula to propositional logic. It allocates a matrix of  $n^2$  boolean variables to each binary relation  $r$ :

$$\begin{matrix} r_{11} & \cdots & r_{1n} \\ \cdots & \cdots & \cdots \\ r_{n1} & \cdots & r_{nn} \end{matrix}$$

where  $r_{ij}$  is true if and only if  $r$  maps atom  $i$  of its domain to atom  $j$  of its range. After having allocated boolean variables to all relations in the formula, the analyzer then proceeds to combining these matrices into matrices of boolean formulas. For example, given a set  $a$ , represented as a degenerate relation,  $a.f$ , the relational image of  $a$  under relation  $f$ , gets the matrix:

$$\begin{array}{c} (a_1 \wedge f_{11}) \vee \cdots \vee (a_n \wedge f_{n1}) \\ \dots \\ (a_1 \wedge f_{1n}) \vee \cdots \vee (a_n \wedge f_{nn}) \end{array}$$

which is the result of matrix multiplication, and states that atom  $i$  is an element of  $a.f$  if and only if there is some atom  $j$  in  $a$  such that  $f$  maps  $j$  to  $i$ . Other Alloy expressions are translated into matrices of boolean formulas in a similar way [10].

To translate formulas, the analyzer combines these matrices into a single propositional formula. For example, the formula  $f = g$ , becomes:

$$(f_{11} \Leftrightarrow g_{11}) \wedge \cdots \wedge (f_{1n} \Leftrightarrow g_{1n}) \wedge \cdots \wedge (f_{n1} \Leftrightarrow g_{n1}) \wedge \cdots \wedge (f_{nn} \Leftrightarrow g_{nn}).$$

which states that variables  $f$  and  $g$  denote the same set of tuples: tuple  $(i, j)$  is in  $f$  if and only if it is in  $g$ .

Given a propositional formula, the analyzer then proceeds to transforming it into conjunctive normal form (CNF) by renaming all subformulas with fresh propositional variables, and conjoining appropriate definitions for these variables to the whole formula [14]. This is done to avoid the exponential blow-up in the size of the formula when it is translated into CNF using distributivity laws. An off-the-shelf SAT solver takes the CNF produced and attempts to find a model. In our case, a satisfying assignment corresponds to a counterexample, which is then output to the user in an appropriate fashion.

## 4 Exploiting Properties of Functions

A field declared in a class is modelled as a relation, but it always maps its object to exactly one other object (or null). Mathematically, this is a function. By exploiting this fact, we can optimize different steps of the analysis presented in the previous section, with the goal of reducing the number of variables and clauses produced in the final CNF, since this will improve the SAT solver's performance.

The main optimization is a representation for functions that requires fewer boolean variables than the representation for general relations. However, this does not reduce the number of variables in the CNF, because the step that translates propositional formulas to CNF adds intermediate variables, and counteracts the benefit of the compact function representation. To harvest this benefit we need two other kinds of optimizations: first a systematic introduction of variables in the first-order formula, and second a series of logical simplifications in the propositional formula. In the next sections, we describe how these optimizations work together to reduce the number of clauses and variables in the CNF.

## 4.1 Function Representation

A relation  $f$  that is a total function maps each atom in its domain to exactly one atom in its range. By representing this atom as an integer in binary form, the encoding of  $f$  requires only  $\lfloor \log(n) \rfloor + 1$  rather than  $n$  boolean variables in each row. From this tighter encoding:

$$\begin{array}{c} f_{11} \cdots f_{1l} \\ \cdots \cdots \cdots \\ f_{n1} \cdots f_{nl} \end{array}$$

we can extract the standard,  $n \times n$  representation:

$$\begin{array}{ccc} \neg f_{11} \wedge \cdots \wedge \neg f_{1l} & \neg f_{11} \wedge \cdots \wedge \neg f_{1(l-1)} \wedge f_{1l} & \cdots \\ \cdots & \cdots & \cdots \\ \neg f_{n1} \wedge \cdots \wedge \neg f_{nl} & \neg f_{n1} \wedge \cdots \wedge \neg f_{n(l-1)} \wedge f_{nl} & \cdots \end{array}$$

where the formula at row  $i$  and column  $j$  is true if and only if row  $i$  in the compact representation of  $f$  represents integer  $j$ . Note that since  $\lfloor \log(n) \rfloor + 1$  bits can represent more than  $n$  values, we must add a side condition that constrains each row of the compact representation in such way that they represent integers less than  $n$ .

If we incorporate this optimization in the Alloy Analyzer, this actually results in an *increase* in the number of variables in the final CNF. This is because the step that transforms propositional logic to CNF counteracts the gain of the compact representation, by renaming all the formulas in the converted matrix with propositional variables. So the resulting CNF has all the variables that it would have had without the compact representation, in addition to all the ones the representation introduces.

## 4.2 Introducing Alloy Variables

To avoid this problem, we first rename all subexpressions that are scalars, i.e. singleton sets, in the first-order formula. Most subformulas that appear in the translation of a fragment of Java code have the form:  $v.f_1 \cdots f_{k_1} = u.g_1 \cdots g_{k_2}$ , where  $a$  and  $b$  are scalars, and  $f_1, \cdots, f_{k_1}$ , and  $g_1, \cdots, g_{k_2}$  are functions, encoding fields. We rename subexpressions of the form  $a.f$  by introducing an Alloy variable  $b$ , and conjoin definitions of the form  $b = a.f$  with the whole formula. Variable  $b$  is a scalar since  $a$  is a scalar and  $f$  is a function. We obtain:  $v_1 = v.f_1 \wedge \cdots \wedge v_{k_1-1} = v_{k_1-2}.f_{k_1-1} \wedge u_1 = u.g_1 \wedge \cdots \wedge u_{k_2} = u_{k_2-1}.g_{k_2} \wedge v_{k_1-1}.f_{k_1} = u_{k_2}$ .

The next section describes logical simplifications that allow a formula of the form  $a.f = b$  to be translated compactly to CNF without adding any additional propositional variables. The CNF's for these formulas are then conjoined. Introducing an Alloy variable for the subexpression  $a.f$  results in  $\lfloor \log(n) \rfloor + 1$  additional boolean variables. If this subexpression were translated to CNF without the introduction of Alloy variables and logical simplifications, it would result in at least  $n^2$  additional boolean variables, since all subformulas are renamed.

### 4.3 Logical Simplifications

We now describe the logical simplifications that allow us to translate  $a.f = b$  compactly to CNF, without introducing any additional propositional variables. They take advantage of the fact that a scalar is represented by a collection of propositional formulas having the property that exactly one of them is true. Informally, the first two simplifications help because they push disjunctions down in the formula's syntax tree. Disjunctions are a source of blow-up when transforming to CNF, and their effect is lessened if they are further away from the root.

**Logical Simplification 1** Consider the formula:

$$(A_1 \wedge B_1) \vee \cdots \vee (A_n \wedge B_n) \quad (1)$$

where  $A_i$  and  $B_i$  ( $1 \leq i \leq n$ ) are boolean formulas. If exactly one of the  $A$  formulas is true, then it can be easily seen that (1) is logically equivalent to:

$$(\neg A_1 \vee B_1) \wedge \cdots \wedge (\neg A_n \vee B_n) \quad (2)$$

**Logical Simplification 2** Consider the formula:

$$((A_1 \wedge B_1) \vee \cdots \vee (A_n \wedge B_n)) \Leftrightarrow C \quad (3)$$

where  $A_i$  and  $B_i$  ( $1 \leq i \leq n$ ) are boolean formulas. If exactly one of the  $A$  formulas is true, then it can be easily seen that (3) is logically equivalent to:

$$(A_1 \wedge (B_1 \Leftrightarrow C)) \vee \cdots \vee (A_n \wedge (B_n \Leftrightarrow C)) \quad (4)$$

Our final simplification is specific to the representation of integers, and relies on the fact that integers can be compared bit by bit.

**Definitions** A *literal* is either a propositional variable, or the negation of one. Given a literal  $a$ , let  $var(a)$  denote the propositional variable corresponding to  $a$ , and  $phase(a)$  be  $+$  ( $-$ ) if  $a$  is  $var(a)$  ( $\neg var(a)$ ).

**Logical Simplification 3** Let  $A_i$  ( $1 \leq i \leq n$ ) be a collection of formulas of the form  $a_1^i \wedge \cdots \wedge a_l^i$ , such that for all  $i, j$ , and for all  $k$  ( $1 \leq k \leq l$ ),  $var(a_k^i) = var(a_k^j)$ , and let  $B_i$  be a similar collection. Consider the formula:

$$A_1 \Leftrightarrow B_1 \wedge \cdots \wedge A_n \Leftrightarrow B_n \quad (5)$$

If exactly one of the  $A_i$  is true, and similarly for the  $B_i$ , and for all  $i$  and  $k$ ,  $phase(a_k^i) = phase(b_k^i)$ , then it can be seen that (5) is logically equivalent to:

$$var(a_1^1) \Leftrightarrow var(b_1^1) \wedge \cdots \wedge var(a_l^1) \Leftrightarrow var(b_l^1) \quad (6)$$

#### 4.4 Theoretical Efficacy

The effect of our optimizations can be predicted theoretically. Let us now compute the number of clauses and variables obtained in the CNF for  $v.f_1 \cdots f_{k_1} = u.g_1 \cdots g_{k_2}$  using our optimizations, to compare them to the case with no optimizations.

In the optimized scheme, each variable is a scalar and is represented with  $l$  boolean variables, where  $l$  denotes  $\lfloor \log(n) \rfloor + 1$ , and  $n$  is the scope, and each function  $f$  is represented with  $nl$  boolean variables. We have seen that  $v.f_1 \cdots f_{k_1} = u.g_1 \cdots g_{k_2}$  can be transformed into a conjunction of  $k_1 + k_2$  formulas of the form  $a.f = b$ . Let  $k$  denote  $k_1 + k_2$ . Since conjunction of CNF can be obtained simply by taking union of clause sets, we can avoid variable introduction and obtain a formula of size  $k\alpha$ , if  $a.f = b$  can be represented with  $\alpha$  clauses.

Consider translating the formula  $a.f = b$  to CNF. Converting  $a$  to the standard  $n \times n$  representation results in a vector of  $n$  elements. We use  $A_i$  to denote the formula on row  $i$ , and similarly for  $b$ . Function  $f$  results in an  $n \times n$  matrix, and we use  $F_{ij}$  to denote the formula at row  $i$ , column  $j$ .

The formula  $a.f = b$  is the conjunction of the following  $n$  formulas:

$$\begin{aligned} (A_1 \wedge F_{11} \vee \cdots \vee A_n \wedge F_{n1}) &\Leftrightarrow B_1 \\ &\quad \wedge \cdots \wedge \\ (A_1 \wedge F_{1n} \vee \cdots \vee A_n \wedge F_{nn}) &\Leftrightarrow B_n \end{aligned}$$

Exactly one of the  $A_i$  is true, so we can apply Logical Simplification 2:

$$\begin{aligned} A_1 \wedge (F_{11} \Leftrightarrow B_1) \vee \cdots \vee A_n \wedge (F_{n1} \Leftrightarrow B_1) \\ &\quad \wedge \cdots \wedge \\ A_1 \wedge (F_{1n} \Leftrightarrow B_n) \vee \cdots \vee A_n \wedge (F_{nn} \Leftrightarrow B_n) \end{aligned}$$

We can then apply Logical Simplification 1:

$$\begin{aligned} (\neg A_1 \vee (F_{11} \Leftrightarrow B_1)) \wedge \cdots \wedge \neg(A_n \vee (F_{n1} \Leftrightarrow B_1)) \\ &\quad \wedge \cdots \wedge \\ (\neg A_1 \vee (F_{1n} \Leftrightarrow B_n)) \wedge \cdots \wedge (\neg A_n \vee (F_{nn} \Leftrightarrow B_n)) \end{aligned}$$

After moving terms around and factoring, we obtain:

$$\begin{aligned} (\neg A_1 \vee (F_{11} \Leftrightarrow B_1)) \wedge \cdots \wedge (F_{1n} \Leftrightarrow B_n) \\ \quad \cdots \\ (\neg A_n \vee (F_{n1} \Leftrightarrow B_1)) \wedge \cdots \wedge (F_{nn} \Leftrightarrow B_n) \end{aligned}$$

Note that for all  $i$ , the formulas  $F_{i1}, \dots, F_{in}$  and  $B_1, \dots, B_n$  satisfy the conditions of Logical Simplification 3. So we apply it to obtain:

$$\begin{aligned} (\neg A_1 \vee (f_{11} \Leftrightarrow b_1) \wedge \cdots \wedge (f_{1l} \Leftrightarrow b_l)) \\ \quad \cdots \\ (\neg A_n \vee (f_{n1} \Leftrightarrow b_1) \wedge \cdots \wedge (f_{nl} \Leftrightarrow b_l)) \end{aligned}$$

Therefore, formula  $a.f = b$  results in  $2nl$  clauses, and no additional intermediate variables. The formula  $v.f_1 \cdots f_{k_1} = u.g_1 \cdots g_{k_2}$  results in  $2nlk$  clauses, and since we added  $k$  variables to break it down, it has  $lk$  intermediate boolean variables.

Consider translating  $v.f_1 \cdots f_{k_1} = u.g_1 \cdots g_{k_2}$  to CNF, without using optimizations. Each subexpression of the form  $a.f$  results in a vector of  $n$  formulas, that are disjunctions of  $n$  conjunctions. For each of these formulas, we introduce  $n$  propositional variables to rename the conjunctions, requiring 3 clauses each for their definitions. We also introduce 1 variable to rename the whole formula, and its definition requires  $n + 1$  clauses. So the subexpression  $a.f$  requires  $n(n + 1)$  additional variables, and  $n(4n + 1)$  clauses. Therefore there are  $n(4n + 1)k$  clauses and  $n(n + 1)k$  variables after the translation of each side of the equality. The equality itself adds  $2n^2$  clauses. We obtain the numbers summarized in the table below.

	Clauses	Intermediate Variables
Non-Optimized	$(4k + 2)n^2 + kn$	$kn^2 + kn$
Optimized	$2kn \lfloor \log(n) \rfloor + k$	$k \lfloor \log(n) \rfloor + k$

## 5 Example

We illustrate our optimizations on an implementation of insertion in red-black trees [4] (Figure 2). The code contains two classes `RBNode` and `RBTree`, and procedure `RBInsert`, which performs insertion into a red-black tree.

We are interested in checking that the red-black invariants are preserved: i.e. that all red nodes must have black children (`Inv1`), and that all paths leading to a node with at most one child have the same number of black nodes (`Inv2`). These invariants maintain a roughly balanced tree.

Figure 2 shows these invariants in Alloy. The function `Inv1` says that for all `RBNodes` `r` that are reachable from the root of `t` by following one or more `left` or `right` fields (`t.root.*(left+right)`), have the property that if `r` is red, then both its children are black. The second property says that for all `RBNodes` `r1` and `r2` reachable from the root, if they both have at most one child<sup>2</sup> (indicated by the calls to function `HasAtMostOneChild`), then the cardinality of the set consisting of all black nodes on the path from `r1` to the root is equal to the cardinality of the corresponding set for `r2`. The symbol `#` denotes set cardinality.

The properties are followed by a series of assertions to be checked. For example, assertion `A` says that for all `RBTrees` `t` and all integers `i`, if the procedure `RBInsert(t,i)` is called and `t` is a well-formed tree (indicated by `Tree(t)`), then the `Inv1` property is preserved. A primed version of a function indicates whether the corresponding property holds in the post-state, in this case the final state of `RBInsert(t,i)`, whereas an unprimed version speaks of the initial state.

<sup>2</sup> In the original algorithm [4], trees have null leaves that are considered to be black.

We do not have these, this is why we need the `HasAtMostOneChild` function.

### Code

```

class RBNode {
    boolean isRed; int key;
    RBNode right; RBNode left;
    RBNode parent;
    public RBNode(int i){
        isRed = false; key = i;
    }
}
class RBTree {
    RBNode root;
    void TreeInsert(RBNode z){
        RBNode k = null;
        RBNode x = this.root;
        while (x != null){
            k = x;
            if (z.key < x.key) x = x.left;
            else x = x.right;}
        z.parent = k;
        if (k == null) this.root = z;
        else if (z.key < k.key) k.left = z;
        else k.right = z;
    }
    static void LeftRotate(RBTree t, RBNode z){
        RBNode y = z.right;
        z.right = y.left;
        if (y.left != null) y.left.parent = z;
        y.parent = z.parent;
        if (z.parent == null) t.root = y;
        else if (z == z.parent.left)
            z.parent.left = y;
        else z.parent.right = y;
        y.left = z;
        z.parent = y;
    }
    static void RBInsert(RBTree t, int i){
        RBNode h = new RBNode(i);
        t.TreeInsert(h);
        h.isRed = true;
        while (h != t.root &&
            h.parent.isRed == true){
            if (h.parent == h.parent.parent.left){
                RBNode y = h.parent.parent.right;
                if (y != null && y.isRed == true){
                    h.parent.isRed = false;
                    y.isRed = false;
                    h.parent.parent.isRed = true;
                    h = h.parent.parent;
                } else {
                    if (h == h.parent.right) {
                        h = h.parent;
                        LeftRotate(t, h); }
                //h.parent.isRed = false; //bug seeded
                h.parent.parent.isRed = true;
                RightRotate(t, h.parent.parent);
            }
        } else { //same as above with
            // left and right inverted
        }
        t.root.isRed = false;
    }
}
}

```

### Specification

```

fun Inv1(t: RBTree) {
    all r: t.root.*(left + right) |
    r.isRed = true => {
        r.right != null => r.right.isRed = false
        r.left != null => r.left.isRed = false
    }
}
fun Inv2(t: RBTree){
    all r1, r2: t.root.*(left + right) {
        HasAtMostOneChild(r1)
        && HasAtMostOneChild(r2) =>
        #{r:RBNode|r in r1.*parent && no r.isRed}
        =
        #{r:RBNode|r in r2.*parent && no r.isRed}
    }
}
fun HasAtMostOneChild(r: RBNode){
    r.left = null || r.right = null
}
assert A { all t: RBTree, i: int |
    RBInsert(t,i) && Tree(t) =>
    Inv1(t) => Inv1'(t)
}
assert B { all t: RBTree, i: int |
    RBInsert(t,i) && Tree(t) =>
    Inv2(t) => Inv2'(t)
}
assert C { all t: RBTree, i: int |
    RBInsert(t,i) && Tree(t) =>
    Inv1(t) && Inv2(t) => Inv1'(t)
}
check C for 5 RBTree,5 RBNode,5 iteration

```

### Counterexample

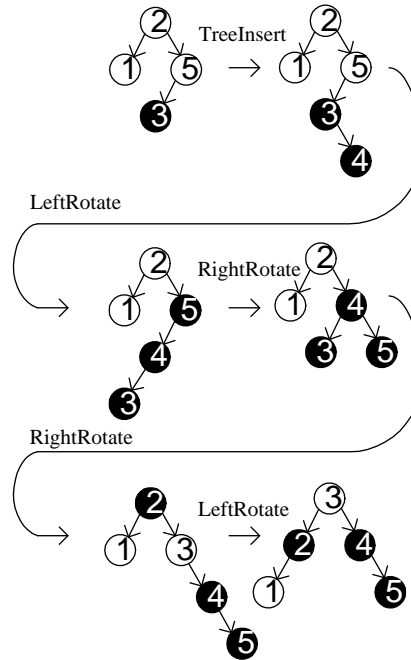


Fig. 2. Red-Black Tree Insertion and its Analysis

assertion	scope	# iter	counter?	time opt	clauses opt	vars opt	time	clauses	vars
A	2	2	no	0	6827	2218	0	15343	7638
	3	3	no	0	14443	4201	0	40042	18727
	4	4	no	3	35642	7887	20	82678	36316
	4	10	no	4	79496	15033	82	188452	81148
	4	20	no	13	152586	26943	—	—	—
	5	5	no	22	58056	13384	232	153259	64575
	5	10	no	43	102281	19854	506	283780	117344
	5	20	no	25	190731	32794	—	—	—
	6	6	no	159	85160	19659	—	—	—
	6	10	no	198	126700	25247	—	—	—
6	20	no	514	230550	39217	—	—	—	
B	2	2	no	0	7066	2310	0	15545	7694
	3	3	no	0	16300	4816	0	41827	19281
	4	4	yes	7	40950	9559	15	87654	37766
	4	10	yes	25	84804	16705	87	193404	82574
	4	20	yes	109	157894	28615	—	—	—
	5	5	yes	12	87369	22109	52	182058	72853
	5	10	yes	44	131594	28579	—	—	—
	5	20	yes	144	220044	41519	—	—	—
	6	6	yes	22	148548	38027	103	318908	121400
	6	10	yes	49	190088	43615	—	—	—
	6	20	yes	132	293938	57585	—	—	—
	7	7	yes	141	240865	62127	—	—	—
	7	10	yes	189	276652	66627	—	—	—
C bug seeded	2	2	no	0	6867	2242	0	15537	7726
	3	3	no	0	15165	4471	0	41172	19162
	4	4	no	2	37886	8679	18	85840	37450
	4	10	no	12	81302	15753	82	192286	82630
	4	20	no	62	153662	27543	—	—	—
	5	5	yes	18	75141	18608	65	172059	70344
	5	10	yes	43	118921	25013	140	303460	123573
	5	20	yes	100	206481	37823	—	—	—
	6	6	yes	77	124936	31383	202	298686	116224
	6	10	yes	82	166056	36915	—	—	—
	7	7	yes	93	199567	50690	—	—	—
	7	10	yes	239	234991	55145	—	—	—
	8	8	yes	272	349823	82336	—	—	—
8	10	yes	641	386261	86216	—	—	—	

Fig. 3. Results

We checked these assertions using a prototype implementation for our analysis, which translates Java code directly to CNF using the optimizations, uses Alloy to translate the specifications to CNF, and conjoins the two. The results are shown in Figure 3, where all times are in seconds. All experiments were run on a 1.1GHz PentiumIII with 640MB of memory, using the BerkMin SAT solver [7]. Some experiments were done after injecting a bug by removing one line in the code (indicated in Figure 2 by the comment `bug seeded`). In Figure 3, dashes indicate either that the experiment took more than 10 minutes or that there was a shortage of memory. The non-optimized experiments are done by translating a subset of Java code to Alloy and uses the Alloy Analyzer equipped with BerkMin as well.

Some of these experiments result in a counterexample. For instance, the counterexample corresponding to assertion C with a bug seeded, for scope of 5 and 5 iterations, is shown in Figure 2. The numbers on each node indicate the

keys, and red nodes are shown in black. The counterexamples goes through 5 iterations to violate assertion 3.

The results show that although the translation without optimization can obtain all the counterexamples very rapidly, which we expected due to the *small scope hypothesis*, the analysis scales better with the optimizations. An empirical study [13] shows that a scope of 6 is enough to obtain full statement and branch coverage for a variety of benchmarks. Our optimizations allow checking all assertions with a scope of 6, and as high as scope of 8 in some cases.

We can also increase the number of iterations to 20 and get an outcome within a minute or two in most cases. For `RBInsert`, 20 iterations for each loop correspond to 1540 lines of code. A state-of-the-art SAT solver can handle formulas having about 250000 clauses in a reasonable amount of time (less than 10 minutes). These experiments show an example of a code fragment of about 1500 lines that can be encoded within the bounds of the SAT solver.

## 6 Related Work

For a bounded instance of a program, our analysis explores *all* the possible inputs and executions, typically accounting for billions of cases. Unlike testing, it can also produce an initial configuration of the heap which leads to a property violation. It differs from finite state verification tools, such as model checking, in that it is modular: procedures may be checked in isolation without requiring a driver. It differs from shape analysis in that it produces sound counterexamples and no false alarms. It also requires no intermediate code annotations, or user-provided abstractions.

*Finite State Verification* FeaVer [9] is a verification tool for C source code, based on the model checker Spin [8]. It extracts a model of a program automatically using a look-up table of abstractions provided by the user. The model is then verified with Spin, which outputs a counterexample when a property is violated. FeaVer has been used successfully to uncover hundreds of bugs in Lucent's Path-Star call processing system.

Bandera [3] is a tool that allows analyzing Java source code with different verification tools. It extracts a finite state model of code using slicing and user-supported data abstraction. The result of the extraction is a model that may be mapped into several model checkers (SMV, Spin) and theorem provers (PVS). Unlike FeaVer and Bandera, our analysis does not provide user-supported abstraction, and this is because it is designed to require as little user intervention as possible.

The Java PathFinder [16] is an environment for checking Java bytecode, that integrates model checking, program analysis, and testing. It allows user-provided abstractions of the program, and uses the Bandera tool for slicing. Although structural properties may be expressed programmatically, Java PathFinder targets event sequences. Moreover Java PathFinder requires an initialization of the heap that fixes it to a particular configuration. Thus it is impossible to have the

tool automatically find an initial configuration that breaks an assertion, as it can be done in our analysis.

The SLAM [1] tool is designed to check if a program obeys API usage rules. It does not require user annotations, and is fully automatic. It abstracts a program into a *boolean program* that is a conservative approximation. The boolean program is then subjected to reachability analysis to see if an error state is attainable. If this is not the case, then there is a guarantee that the original program cannot reach the error state. If an error state is reachable, then it is analyzed automatically to see if it is part of a feasible execution, in which case a counterexample is output to the user. If no feasible execution leads to the error state, then appropriate predicates are added to the abstraction, again automatically. The process then starts over with this refined abstract program. SLAM targets temporal safety properties, and not structural properties.

*Shape Analysis* Shape analysis algorithms [15] can identify invariants for programs that manipulate heap-allocated storage. They represent the heap as *shape graphs*, conservative abstractions that capture properties at different points in the program. The parametric shape analysis (PSA) [15] method uses a 3-valued logic to represent shape graphs, and is a framework that can be instantiated with different *instrumentation predicates* that retain more refined information about concrete heaps, and can help to identify different classes of properties. PSA differs from our method in that it can either prove a property or is inconclusive, whereas we output a counterexample or “inconclusive”. PSA does not generate concrete counterexamples. Moreover, because of the abstractions used, it can output spurious error reports. Finally, PSA does not seem to scale very well and requires instrumentation predicates that tailor the analysis for the discovery of particular properties. In our method the translation of the code is independent of the property to verify, and does not require property-specific abstractions.

*Theorem Proving* The Extended Static Checker [5] uses a powerful theorem prover to check code against user-specified specifications. Structural properties such as those handled in our analysis are not expressible. Experience has shown that ESC requires many intermediate code annotations, making it less practical. An extension of ESC, the VeriFun tool [6], uses predicate abstraction, powerful decision procedures, and automated successive refinement. It requires no user annotation beyond the property to be checked. It differs with our analysis in that it cannot readily handle the kinds of structural properties we consider here.

## 7 Conclusions

We presented a suite of optimizations that results in much improved scalability for our analysis of object-oriented code, which targets structural properties of the heap, requires no user-annotation, and outputs no false alarms. Our optimizations are a suite of simple but judiciously chosen logical simplifications, that are not incremental, i.e. their *composition* results in an improvement in scalability.

A conventional way to scale an analysis such as this, is to require user-provided specifications for all procedures. The ability to handle longer code sequences allows a longer procedure to be considered, and for smaller procedures, it allows specifications to be omitted. The scalability of this analysis is therefore crucial to allow checking code fragments in which specifications are written at a coarser, more economical granularity.

Our experimental results use a prototype tool that translates a subset of Java directly to CNF using the optimizations, while the non-optimized tool translated to Alloy, and therefore benefitted from the Alloy Analyzer's internal simplifications. As part of future work, we plan to incorporate our optimizations in the Alloy Analyzer, so that we can benefit from its simplifications as well. We also plan to run more experiments on different code bases, to further demonstrate the effect of our optimizations.

## References

1. T. Ball and S. R. Rajamani. "Boolean Programs: A Model and Process for Software Analysis", MSR Technical Report 2000-14, Microsoft Research, February 2000.
2. D. R. Chase, M. Wegman and F. Zadeck. "Analysis of Pointers and Structures", *Proc. Conf. on Programming Language Design and Implementation*, 1990.
3. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, H. Zheng. "Bandera: Extracting Finite-State Models from Java Source Code", *Proc. International Conference on Software Engineering*, June 2000.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest. "Introduction to Algorithms", MIT Press, 1990.
5. D. Detlefs, K. R. Leino, G. Nelson, and J. Saxe. "Extended Static Checking". Technical Report 159, Compaq Systems Research Center, 1998.
6. Cormac Flanagan. Personal communication.
7. E. Goldberg and Y. Novikov. "BerkMin: A fast and robust SAT-solver", *In Design, Automation, and Test in Europe*, March 2002.
8. G.J. Holzmann. "The Model Checker Spin", *IEEE Trans. on Software Engineering*, Vol. 23, 5, May 1997.
9. G. J. Holzmann and M. H. Smith. "Automating Software Feature Verification", *Bell Labs Technical Journal*, Vol. 5, 2, April-June 2000.
10. Daniel Jackson. "Automating First-Order Relational Logic", *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, San Diego, November 2000.
11. D. Jackson, I. Shlyakhter and M. Sridharan. "A Micromodularity Mechanism", *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, 2001.
12. D. Jackson and M. Vaziri. "Finding Bugs with a Constraint Solver", *Proc. International Conference on Software Testing and Analysis*, August 2000.
13. A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. "Evaluating the Small Scope Hypothesis", MIT Laboratory for Computer Science, Submitted for publication, September 2002.
14. D. A. Plaisted and S. Greenbaum. "A Structure-Preserving Clause Form Translation", *Journal of Symbolic Computation*, 2:293-304, 1986.
15. M. Sagiv, T. Reps, and R. Wilhelm. "Parametric Shape Analysis via 3-Valued Logic", *Proc. ACM Symposium on Principles of Programming Languages*, 1999.
16. W. Visser, K. Havelund, G. Brat and S. Park. "Model Checking Programs", *International Conference on Automated Software Engineering*, September 2000.