Analysis of Invariants for Efficient Bounded Verification

Juan P. Galeotti, Nicolás Rosner,
Carlos G. López Pombo
Department of Computer Science, FCEyN
Universidad de Buenos Aires, Argentina
{jgaleotti,nrosner,clpombo}@dc.uba.ar

Marcelo F. Frias
Department of Software Engineering,
Buenos Aires Institute of Technology (ITBA)
Argentina
mfrias@itba.edu.ar

ABSTRACT

SAT-based bounded verification of annotated code consists of translating the code together with the annotations to a propositional formula, and analyzing the formula for specification violations using a SAT-solver. If a violation is found, an execution trace exposing the error is exhibited. Code involving linked data structures with intricate invariants is particularly hard to analyze using these techniques.

In this article we present TACO, a prototype tool which implements a novel, general and fully automated technique for the SAT-based analysis of JML-annotated Java sequential programs dealing with complex linked data structures. We instrument code analysis with a symmetry-breaking predicate that allows for the parallel, automated computation of tight bounds for Java fields. Experiments show that the translations to propositional formulas require significantly less propositional variables, leading in the experiments we have carried out to an improvement on the efficiency of the analysis of orders of magnitude, compared to the non-instrumented SAT-based analysis. We show that, in some cases, our tool can uncover bugs that cannot be detected by state-of-the-art tools based on SAT-solving, model checking or SMT-solving.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.2.1 [**Software Engineering**]: Specifications; D.2.4 [**Software Engineering**]: Program verification—Class invariants, programming by contract, formal methods.

General Terms

Verification, Languages

Keywords

Static analysis, SAT-based code analysis, Alloy, KodKod, DynAlloy.

ISSTA'10, July 12–16, 2010, Trento, Italy.

1. INTRODUCTION

SAT-based analysis of code allows one to statically find failures in software. This requires appropriately translating the original piece of software, as well as some assertion to be verified, to a propositional formula. The use of a SAT-solver then allows one to find a valuation for the propositional variables that encodes a failure: a valid execution trace of the system that violates the given assertion. With variations, this is the approach followed by CBMC [6], Saturn [32] and F-Soft [17] for the analysis of C code, and by Miniatur [11] and JForge [9] for the analysis of Java code.

In the presence of contracts for invoked methods, modular SAT-based analysis can be done by first replacing the calls in a method by the corresponding contracts and then analyzing the resulting code. This is the approach followed for instance in [9]. One important limitation remains at the *intraprocedural* level, where the code for a single method (already including the contracts or the inlined code for called methods) has to be analyzed. Code involving linked data structures with rich invariants (such as circular lists, red-black trees, AVL trees or binomial heaps) is hard to analyze using these techniques.

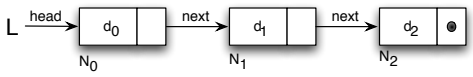
In this article we present TACO (Translation of Annotated COde), our prototype tool implementing a novel, general and fully automated technique for SAT-based analysis of sequential annotated code involving complex linked data structures. This technique relies on a novel and effective way of removing variables in the translation to a propositional formula. In Section 4 we will present experimental results showing that the technique we present shows significant improvements in SAT-based intraprocedural program analysis and allows us to uncover bugs that could not be detected using state-of-the-art tools based on model checking or SMT-solving.

To describe the technique at a high level of abstraction let us consider the following class for singly-linked structures:

```
public class List {
  LNode head;
}
```

```
public class LNode {
  LNode next;
  int key;}
```

Assuming that we use this structure for representing a singly linked list, we require lists to be acyclic. Let us also assume that nodes have identifiers $N_0, N_1, N_2, \ldots$, and that nodes are kept in the list in order according to their identifiers ($N_0 < N_1 < \cdots$). Thus, a list instance will have the shape



The model we will adopt of the memory heap (presented originally in [19]), models Java fields as functions from object identifiers to object identifiers or the value `null` in the codomain. For instance, if we have in the `LNode` domain 3 nodes whose identifiers are N_0, N_1 and N_2, field `next` will be modeled as a total function

$$\texttt{next} : \{N_0, N_1, N_2\} \rightarrow \{N_0, N_1, N_2, \texttt{null}\} \ .$$

In the translation to a propositional model, field `next` is modeled as a matrix of propositional variables

	N_0	N_1	N_2	`null`
N_0	p_{N_0,N_0}	p_{N_0,N_1}	p_{N_0,N_2}	$p_{N_0,\texttt{null}}$
N_1	p_{N_1,N_0}	p_{N_1,N_1}	p_{N_1,N_2}	$p_{N_1,\texttt{null}}$
N_2	p_{N_2,N_0}	p_{N_2,N_1}	p_{N_2,N_2}	$p_{N_2,\texttt{null}}$

For instance, propositional variable p_{N_1,N_2} models whether $\texttt{next}(N1) = N2$ or not. Notice that since node identifiers are assumed to be contiguous, certain variables are deemed to be false. In this case, variables p_{N_0,N_2}, p_{N_1,N_0}, p_{N_2,N_0} and p_{N_2,N_1} must be false in all models. Since the invariant asks for lists to be acyclic, variables p_{N_0,N_0}, p_{N_1,N_1} and p_{N_2,N_2} are always false as well. If we could determine this before translating to a propositional formula, then these 7 variables could be removed in the translation process and be replaced by the truth value *false*. This would lead to a formula with fewer variables, and therefore to a simpler SAT problem that often can be solved in a fraction of the original analysis time.

The contributions are summarized as follows:

1. We present a novel and fully automated technique for canonicalization of the memory heap in the context of SAT-solving, which assigns identifiers to heap objects in a well-defined manner (to be made precise in Section 3.1).

2. Using this ordering, we present a fully automated and parallel technique for determining which variables can be removed. The technique consists of computing bounds for Java fields (to be defined in Section 3.2). The algorithm only depends on the invariant of the class under analysis. Therefore, the computed bounds can be reused across all the analyses in a class, and the cost of computing the bounds can be amortized.

3. We present several case studies using complex linked data structures showing that the technique improves the analysis by reducing analysis times by several orders of magnitude in case correct code is analyzed. We also show that the technique can efficiently discover errors seeded using mutant generation [7]. Finally, we report on a previously unknown [31] error found in a benchmark presented in [30]. This error was not detected by several state-of-the-art tools based on SAT-solving, model checking or SMT-solving.

4. We discuss to what extent the techniques presented in this article can be used by related tools.

The article is organized as follows. In Section 2 we describe the translation of JML-annotated sequential Java code to a SAT problem. In Section 3 we present our novel technique for program analysis. In Section 4 we present the experimental results. In Section 5 we discuss related work. Finally, in Section 6 we discuss lines for further work and draw conclusions about the results presented in the article.

2. FROM JML TO SAT

In this section we present an outline of our translation of JML [12] annotated Java code to a SAT problem. The translation is in intention not much different from translations previously presented by other authors [10] or by some of the authors of this article [15]. A schematic description of TACO's architecture that shows the different stages in the translation process is provided in Fig. 1. In order to simplify writing properties of linked structures we use in this article an extension of JML with a construct `\reach(l, T, [f1,...,fk])` denoting the set of objects of type `T` reachable from a location `l` using fields `f1,...,fk`.

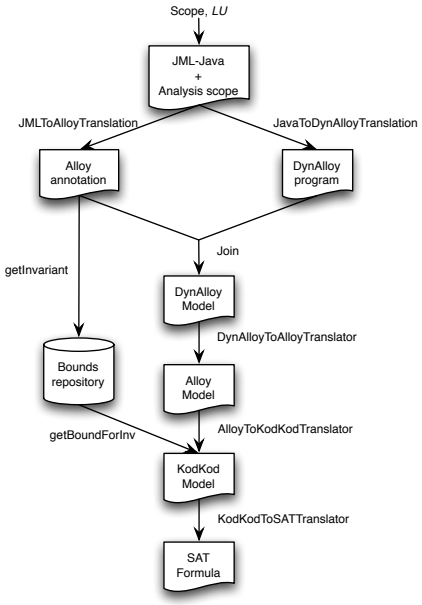


Figure 1: Translating annotated code to SAT.

Our translation uses Alloy [18] as an intermediate language. This is an appropriate decision because Alloy is close to JML, and the Alloy Analyzer [18] provides a simple interface to several SAT-solvers. Also, Java code can be translated to DynAlloy programs [15]. DynAlloy [13] is an extension of Alloy that allows us to specify actions that modify the state much the same as Java statements do. Action behavior is specified by pre and post conditions given as Alloy formulas. From these *atomic* actions we build complex DynAlloy programs modeling sequential Java code.

As shown in Fig. 1 the analysis receives as input an annotated method, a *scope* bounding the sizes of object domains, and a bound LU for the number of loop iterations. JML annotations allow us to define a method contract (using constructs such as `requires`, `ensures`, `assignable`, `signals`, etc.), and invariants (both static and non-static). A contract may include normal behavior (how does the system behave when no exception is thrown) and exceptional behavior (what is the expected behavior when an exception is thrown). The scope constrains the size of data domains during analysis. For example, if we are analyzing a model for singly linked lists holding objects of type Data, the scope constrains the number of List objects, LNode objects and Data objects to be used during analysis (for instance 1 List, 10 LNode, 10 Data is a plausible scope). This is a restriction on the precision of the analysis. If an analysis does not find a bug, it means no bug exists *within* the provided scope for data domains. Bugs could be found repeating the

analysis using larger scopes. Therefore, only a portion of the program domain is actually analyzed. Fortunately, this is sufficient to expose many failures, since failures can often be reproduced with few data [1].

The annotations are then translated to Alloy formulas using translation `JMLtoAlloyTranslation` [15], and the method under analysis is translated to a DynAlloy program using translation `JavaToDynAlloyTranslation` [15]. The resulting translations are joined into a single DynAlloy model that includes a partial correctness assertion. The assertion states that every terminating execution of the code starting in a state satisfying the precondition and the class invariant leads to a final state that satisfies the postcondition and preserves the invariant.

In order to handle loops we constrain the number of iterations by performing a user-provided number of loop unrolls LU. Therefore, the (static) analysis will only find bugs that could occur performing up to LU iterations at runtime. Notice that an interaction occurs between the scope and LU. This is a natural situation under these constraints, and similar interactions occur in other tools such as Miniatur [11] and JForge [9].

As shown in Fig. 1, DynAlloy models are translated to Alloy models using the `DynAlloyToAlloyTranslator`. We will not focus on this translation, which has already been discussed in [14], but rather emphasize the way Java classes are modeled in Alloy as a result of applying the translations. This will allow us to show how the technique we will present in Section 3 fits in the code analysis process. For the Java class for lists in Section 1, the resulting Alloy model includes the following type definitions:

```
one sig null {}

sig List {                      sig LNode {
  head : LNode + null }           next : LNode + null,
                                  key : Int }
```

According to Alloy semantics, signatures define sets of atoms. These atoms are akin to the identifiers we mentioned in Section 1. The modifier `one` in signature `null` constrains the signature to have a single datum. Signature `List` defines list atoms and also includes a signature field `head`. Field `head` denotes a total function from `List` atoms to `LNode` atoms or `null` (in Alloy notation, `head : List -> one (LNode+null)`). Similarly, we have `next : LNode -> one (LNode+null)`. Given scopes s for signature `S` and t for signature `T`, one can determine the number of propositional variables required in order to represent a field `f : S -> one (T+null)` in the SAT model. Notice that `S` and `T` will contain atoms $S_1, \ldots, S_s$ and $T_1, \ldots, T_t$, respectively. We will use a matrix $M_{\mathtt{f}}$ holding $s \times (t+1)$ propositional variables to represent field `f`:

$M_{\mathtt{f}}$	T_1	T_2	$\ldots$	T_t	`null`
S_1	p_{S_1,T_1}	p_{S_1,T_2}	$\cdots$	p_{S_1,T_t}	$p_{S_1,\mathtt{null}}$
S_2	p_{S_2,T_1}	p_{S_2,T_2}	$\cdots$	p_{S_2,T_t}	$p_{S_2,\mathtt{null}}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
S_s	p_{S_s,T_1}	p_{S_s,T_2}	$\cdots$	p_{S_s,T_t}	$p_{S_s,\mathtt{null}}$

Intuitively, a variable p_{S_i,T_j} $(1 \leq i \leq s, 1 \leq j \leq t)$ models whether the pair of atoms/identifiers $\langle S_i, T_j \rangle$ belongs to `f` or, equivalently, whether $S_i.\mathtt{f} = T_j$. A variable $p_{S_i,\mathtt{null}}$ models whether $S_i.\mathtt{f} = \mathtt{null}$. Actually, as shown in Fig. 1, Alloy models are not directly translated to a SAT problem, but to the intermediate language KodKod [27].

A distinguishing feature of KodKod is that it enables the prescription of partial instances in models. Indeed, each Alloy 4 field `f` is translated as described above, together with two *bounds* (relation instances) $L_{\mathtt{f}}$ (the lower bound) and $U_{\mathtt{f}}$ (the upper bound). These bounds provide useful information. Consider for instance relation `next`. If a tuple $\langle N_i, N_j \rangle \notin U_{\mathtt{next}}$, then no instance of field `next` can contain $\langle N_i, N_j \rangle$, allowing us to replace p_{N_i,N_j} in $M_{\mathtt{next}}$ (the matrix of propositional variables associated with relation `next`) by the truth value *false*. Similarly, if $\langle N_i, N_j \rangle \in L_{\mathtt{next}}$, pair $\langle N_i, N_j \rangle$ must be part of any instance of field `next` (allowing us to replace variable p_{N_i,N_j} by the truth value *true*). Thus, the presence of bounds allows us to determine the value of some entries in the KodKod representation of a given Java field. Thus, bounds contribute in the translation by removing propositional variables. Since, in the worst case, the SAT-solving process grows exponentially with respect to the number of propositional variables, getting rid of variables often improves (as we will show in Section 4) the analysis time significantly. In our example, determining that a pair of atoms $\langle N_i, N_j \rangle$ can be removed from the bound $U_{\mathtt{next}}$ allows us to remove a propositional variable in the translation process. When a tuple is removed from an upper bound, the resulting bound is said to be *tighter* than before.

Notice that the translation from Java code to a SAT problem could be implemented as a one-step transformation. In this sense, the translation just described does not depend on Alloy, DynAlloy or KodKod and can be used in more general settings. Yet these languages and their supporting tools offer useful infrastructure to prototype the translation.

In Section 3 we concentrate on how to determine if a given pair can be removed from an upper bound relation and therefore produce tighter upper bounds.

3. COMPUTING TIGHT BOUNDS

In this section we present the main contributions of the article. Both the symmetry breaking predicates to be introduced in Section 3.1 and the algorithm for automated computation of tight upper bounds presented in Section 3.2 are novel contributions of this article.

Up to this point in the article we have made reference to three different kinds of bounds, namely:

- The bounds on the size of data domains used by the Alloy Analyzer. Generally, these are referred to as *scopes* and should not be confused with the intended use of the word *bounds* in this section.
- In DynAlloy, besides imposing scopes on data domains as in Alloy, we bound the number of loop unrolls. Again, this bound is not to be confused with the notion of bound that we will use in this section.
- In the end of Section 2 we made reference to the lower and upper bounds ($L_{\mathtt{f}}$ and $U_{\mathtt{f}}$) attached to an Alloy field `f` during its translation to a KodKod model. In this section, we use the term *bound* to refer to the upper bound $U_{\mathtt{f}}$.

Complex linked data structures usually have complex invariants that impose constraints on the topology of data and on the values that can be stored. For instance, a red-black tree, a variety of balanced, ordered binary tree, requires that:

1. For each node n in the tree, the keys stored in nodes

in the left subtree of n are always smaller than the key stored in n. Similarly, keys stored in nodes in the right subtree are always greater than the key stored in n.

2. Nodes are colored red or black, and the tree root is colored black.

3. In any path starting from the root node there are no two consecutive red nodes.

4. Every path from the root to a leaf node has the same number of black nodes.

In the Alloy model result of the translation, Java fields are mapped to total functional relations. For instance, field *left* is mapped to a total functional relation. Suppose that we are interested in enumerating instances of red black trees that satisfy a particular predicate. This predicate could be the above representation invariant, or a method precondition involving red black trees. Let us assume it is the above invariant. Furthermore, let us assume that:

1. nodes come from a linearly ordered set, and

2. trees have their node identifiers chosen in a canonical way (for instance, a breadth-first order traversal of the tree yields an ordered listing of the node identifiers).

That is, given a tree composed of nodes $N_0, N_1, \ldots, N_k$, node N_0 is the tree root, $N_0.left = N_1$, $N_0.right = N_2$, etc. Notice that the breadth-first ordering already imposes some constraints. For instance, it is not possible that $N_0.left = N_2$. Moreover, if there is a node to the left of node N_0, it *has* to be node N_1 (otherwise the breadth-first listing of nodes would be broken). At the Alloy level, this means that $\langle N_0, N_2\rangle \in$ `left` is infeasible, and the same is true for $N_3, \ldots, N_k$ instead of N_2. Recalling the discussion at the end of Section 2, this means that we can get rid of several propositional variables in the translation of the Alloy encoding of the invariant to a propositional SAT problem. Actually, as we will show in Section 4, for a scope of 10 tree nodes, this analysis allows us to reduce the number of propositional variables required to model the invariant state from 650 to 200.

The usefulness of the previous reasonings strongly depends on:

1. being able to guarantee, fully automatically, that nodes are placed in the heap in a canonical way, and

2. being able to automatically determine, for each class field `f`, what are the infeasible pairs of values that can be removed from the bound $U_{\mathtt{f}}$.

A predicate that reduces the number of equivalent models by inducing a canonical ordering in the heap nodes is a "symmetry breaking" predicate. We will present an appropriate symmetry breaking predicate in Section 3.1. In Section 3.2 we present a fully automatic and effective technique for checking infeasibility.

3.1 A New Predicate for Symmetry Breaking

In order to describe predicates concisely we will use Alloy notation, which is thoroughly described in [18]. Alloy is a relational language. Terms are built from signature names (which stand for unary relations –sets), from signature fields (binary relations in the case of fields coming from Java code), and from typed variables denoting atoms from the corresponding signature. There are three constants in the language: *univ* (which denotes the set of all atoms in the universe), *none* (that denotes the empty set), and *iden* (which denotes the binary identity relation over the atoms in *univ*). If T is a term that denotes a binary relation, then $\sim T$, $*T$ and $\hat{}\,T$ denote transposition, reflexive-transitive closure and transitive closure of the relation denoted by T, respectively. Union of relations is noted as $+$, intersection as &, difference as $-$, and sequential composition as ".". For instance, the expression `head.*next` relates each input list to the nodes in the list or the value *null* if the list is acyclic. From terms we build atomic formulas T_1 in T_2 or $T_1 = T_2$ stating that relation T_1 is contained in relation T_2, and that T_1 and T_2 are the same relation, respectively. From atomic formulas we build complex formulas using the connectives ! (negation), && (conjunction), || (disjunction) and => (implication). Existentially quantified formulas have the form "some $x : S \mid \alpha$", where x ranges over the elements in signature S, and α is a formula. Similarly, universally quantified formulas have the form "all $x : S \mid \alpha$". For a term T, formula "`no` T" states that the relation denoted by T is empty.

The following Alloy predicate

```
pred acyclic[l : List] { all n : LNode |
   n in l.head.*next => n !in n.^next }
```

describes acyclic lists. Running the predicate in the Alloy Analyzer using the command

```
run acyclic for exactly 6 Object, exactly 1 List,
                exactly 4 LNode, exactly 1 Data
```

yields lists that are permutations (on signature LNode). Actually, any permutation of LNode that stores data in the same order as any of these lists, is also a model. Pruning the state space by removing permutations on signature LNode contributes to improving the analysis time. For singly linked lists, a predicate forcing nodes to be used in the order LNode0 → LNode1 → LNode2 → ... removes symmetries.

The idea of canonicalizing the heap in order to reduce symmetries is not new. In the context of explicit state model checking, the articles [16, 23] present different ways of canonicalizing the heap ([16] uses a depth-first search traversal, while [23] uses a breadth-first search traversal of the heap). The canonicalizations require modifying the state exploration algorithms, and involve computing hash functions in order to determine the new location for heap objects in the canonicalized heap. Notice that:

- The canonicalizations are given algorithmically (which is not feasible in a SAT-solving context).

- Computing a hash function requires operating on integer values, which is appropriate in an algorithmic computation of the hash values, but is not amenable for a SAT-solver.

In the context of SAT-based analysis, [21] proposes to canonicalize the heap, but the canonicalizations have to be provided by the user as ad-hoc predicates depending on the invariants satisfied by the heap.

In this section we present a novel family of predicates that canonicalize arbitrary heaps. In order to include the predicates we will instrument the Alloy model obtained by the translation from the annotated source code. The predicates are automatically instantiated in these Alloy models. In order to make the presentation of the symmetry breaking

```
sig Object { }
one sig null, Red, Black {}
sig RBTree extends Object {root : RBTNode + null}
sig RBTNode extends Object {
   left : RBTNode + null,
   value : Data + null,
   color : Red + Black,
   right : RBTNode + null }
sig Data extends Object { }
```

Figure 2: An Alloy model for red-black trees.

predicates more amenable, we will do it through a running example. Let us consider the Alloy model for red-black trees presented in Fig. 2. Keyword `extends` constrains the extending signature to be a subset of the extended one. We will use run or check commands in the Alloy Analyzer whose scopes will be:

```
exactly 1 RBTree, exactly 5 RBTNode, exactly 5 Data
```

Our model of Java heaps consists of graphs $\langle N, E, L, R\rangle$ where N (the set of heap nodes), is a set comprising elements from signature Object and appropriate value signatures (int, String, etc.). E, the set of edges, contains pairs $\langle n_1, n_2\rangle \in N \times N$. L is the edge labeling function. It assigns class field names to edges. An edge between nodes n_1 and n_2 labeled f_i means that $n_1.f_i = n_2$. The typing must be respected. $R \subseteq N$ is the set of heap root nodes (method arguments and static class fields, of object type).

In our running example, nodes are the objects from signatures `RBTree`, `RBTNode` and `Data`, or the value `null`. Labels correspond to field names in the model, and the root is the argument `this`, of type `RBTree`.

We then instrument the Alloy model as follows. If the scope for signature T is k, we include singletons $T_0, \ldots, T_{k-1}$. For the trees example we have:

```
one sig RBTree0 extends RBTree {}
one sig RBTNode0,...,RBTNode4 extends RBTNode {}
one sig Data0,...,Data4 extends Data {}
```

We also introduce auxiliary functions. Function $nextT$ establishes a linear order between elements of type T. Function $minT$ returns the least object (according to the ordering $nextT$) in an input subset. Function $prevsT$ returns the nodes in signature T smaller than the input parameter. For the example (only for signature `RBTNode`), we have:

```
fun nextRBTNode[] : RBTNode -> lone RBTNode {
   RBTNode0->RBTNode1 + RBTNode1->RBTNode2 +
   + RBTNode2 -> RBTNode3 + RBTNode3->RBTNode4 }

fun minRBTNode [ns: set RBTNode] : lone RBTNode {
   ns - ns.^(nextRBTNode[]) }

fun prevsRBTNode[n : RBTNode] : set RBTNode {
   n.^(~nextRBTNode[]) }
```

Each recursive field `r : S -> (S + null)` from signature S is split into two *partial* functions (thus the `lone` modifier) `fr : S -> lone (S + null)` (the forward part of the field, mapping nodes to strictly greater nodes or `null`) and `br : S -> lone S` (the backward part of the field). Non-recursive fields are not modified. In our example the resulting fields are:

```
root   : RBTree -> one (RBTNode + null),
fleft  : RBTNode -> lone (RBTNode + null),
bleft  : RBTNode -> lone RBTNode,
fright : RBTNode -> lone (RBTNode + null),
bright : RBTNode -> lone RBTNode,
value  : RBTNode -> one (Data + null).
```

Java fields must be total functions. We add new facts stating that for each recursive field r_i, the domains of fr_i and br_i form a partition of r_i's domain, making $fr_i + br_i$ a well–defined total function. For our example we have:

```
fact { no ((fleft.univ) & (bleft.univ)) and
       RBTNode = fleft.univ + bleft.univ and
       no ((fright.univ) & (bright.univ)) and
       RBTNode = fright.univ + bright.univ }
```

The instrumentation we are presenting will require us to talk about the reachable heap objects. Since all the objects will be reachable through forward fields, we obtain a more economic (regarding the translation to a propositional formula) description of the reachable heap objects using forward fields. In our example, instead of using the expression

```
this.*(root + left + value + right)
```

to characterize the reachable heap objects, we will use the expression `FReach` defined as

```
this.*(root + fleft + value + fright).
```

We now introduce facts forcing the SAT-solver to choose nodes in a canonical order. Intuitively, we will order heap nodes by looking at their parents in the heap. A node may have no parents (in case it is a heap root), or have several parent nodes. In the latter case, among the parents we will distinguish the minimal one (according to a global ordering) and will call it the *min parent* of n (denoted $minP[n]$). For the example, functions `globalMin` and `minP` are defined as follows:

```
fun globalMin[s : set Object] : Object {
  s - s.^(RBTree0->RBTNode0 + RBTNode0->RBTNode1 +
  ...+ RBTNode3->RBTNode4 + RBTNode4->Data0 +
  Data0->Data1 +...+ Data3->Data4)}

fun minP[o : Object] : Object {
  globalMin[(root + fleft + value + fright).o] }
```

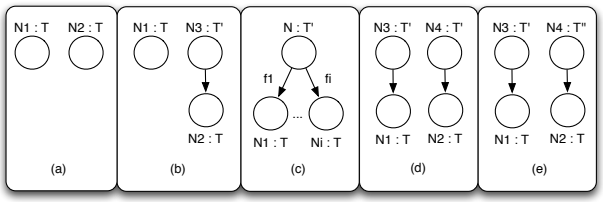


Figure 3: Comparing nodes using their min-parents.

In order to determine how to order any pair of nodes from the same signature T, we will consider the possibilities depicted in Fig. 3. The ordering is then characterized by the following conditions:

(a) To sort two root nodes of type T, we use the ordering in which the formal parameters or static fields were declared in the source Java file. Then a fact is added including the explicit ordering information.

(b) To sort a root node and a non-root node of the same type, we add a fact stating that root nodes are always smaller than non root nodes.

(c) To sort nodes $N_1, \ldots, N_i$ of the same type such that $\texttt{minP}[N_1] = \ldots = \texttt{minP}[N_i] = N$, notice that since Java fields are functions, there must be i different fields $f_1, \ldots, f_i$ such that $N.f_1 = N_1$, $N.f_2 = N_2$, etc. We then use the ordering in which the fields were declared in the source Java file to sort $N_1, \ldots, N_i$.

(d) Let N_1 (with min parent N_3) and N_2 (with min parent N_4) be nodes of the same type. If N_3 and N_4 are distinct and have the same type, then we will sort N_1 and N_2 following the order between N_3 and N_4.
(e) Finally, in order to sort nodes N_1 and N_2 of type T whose min parents have different type, we will use the ordering in which the classes of the parent nodes were defined in the source Java file.

Since in the running example there is exactly one root node (namely, `this`), we statically and automatically determine it is not necessary to add a fact for condition (a). Since there is exactly one object in class `RBTree`, we do not add a fact for condition (b). Regarding condition (c), since there is only one field from signature `RBTree` to signature `RBTNode`, there are no two objects with type `RBTNode` with the same min parent in signature `RBTree`.

Fact "orderRBTNodeCondition(c)" below orders objects of type `RBTNode` with the same min parent of type `RBTNode`:

```
fact orderRBTNodeCondition(c){
 all disj o1, o2 : RBTNode |
  let a = minP[o1] | let b = minP[o2] |
   (o1+o2 in FReach and some a and some b and
   a = b and a in RBTNode and o1 = a.fleft and
   o2 = a.fright) implies o2 = o1.nextRBTNode[]}
```

Fact "orderRBTNodeCondition(d)" orders objects of type `RBTNode` with different min parents of type `RBTNode`. A similar fact is necessary in order to sort objects of type `Data` with different `RBTNode` min parents.

```
fact orderRBTNodeCondition(d) {
 all disj o1, o2 : RBTNode |
  let a = minP[o1] | let b = minP[o2] |
   (o1+o2 in FReach and some a and some b and
   a!=b and a+b in RBTNode and a in prevsRBTNode[b])
       implies o1 in prevsRBTNode[o2]}
```

Regarding condition (e), fact "orderRBTNodeCondition(e)" orders objects of type `RBTNode` whose min parents are one of type `RBTree`, and the other of type `RBTNode`:

```
fact orderRBTNodeCondition(e){
  all disj o1, o2 : RBTNode |
    let a = minP[o1] | let b = minP[o2] |
      (o1+o2 in FReach and some a and some b and
        a in RBTree and b in RBTNode)
          implies o1 in prevsRBTNode[o2]}
```

In order to avoid "holes" in the ordering, for each signature T we add a fact "compactT" stating that whenever a node of type T is reachable in the heap, all the smaller ones in the ordering are also reachable. For signature `RBTNode` we have:

```
fact compactRBTNode { all o : RBTNode | o in FReach
    implies prevsRBTNode[o] in FReach}
```

Finally, the instrumentation modifies the facts, functions, predicates and assertions of the original model by replacing each occurrence of a recursive field r_i with the expression $fr_i + br_i$. For instance, if a fact *acyclicRBTree* is used to state that trees are acyclic structures:

```
fact acyclicRBTree { all t : RBTree, n : RBTNode |
   n in t.root.*(left + right) implies
      n !in n.^(left + right) }
```

in the instrumented model it is replaced by the fact

```
fact acyclicRBTree { all t : RBTree, n : RBTNode |
  n in t.root.*(fleft+bleft+fright+bright)
    implies n !in n.^(fleft+bleft+fright+bright) }
```

The following theorems show that the instrumentation is correct.

THEOREM 3.1. *Given a heap H for a model, there exists a heap H' isomorphic to H and whose ordering between nodes respects the instrumentation. Moreover, if an edge $\langle n_1, n_2\rangle$ is labeled r (with r a recursive field), then: if n_1 is smaller (according to the ordering) than n_2 (or n_2 is `null`), then $\langle n_1, n_2\rangle$ is labeled in H' fr. Otherwise, it is labeled br.*

Theorem 3.1 shows that the instrumentation does not miss any bugs during code analysis. If a counterexample for a partial correctness assertion exists, then there is another counterexample that also satisfies the instrumentation.

THEOREM 3.2. *Let H, H' be heaps for an instrumented model. If H is isomorphic to H', then $H = H'$.*

Theorem 3.2 shows that the instrumentation indeed yields a canonicalization of the heap.

3.2 Symmetry Breaking and Tight Bounds

In this section we present the main result of this article. It is interesting to notice that despite the symmetry breaking predicates that come with the standard distribution of the Alloy Analyzer, isomorphic models are generated. While in the original Alloy model functions *left* and *right* are each encoded using $n \times (n + 1)$ propositional variables, due to the canonical ordering of nodes we can remove arcs from relations. In order to determine whether edges $N_i \rightarrow N_j$ can be part of field `F` or can be removed from U_F, TACO proceeds as follows:

1. Synthesizes the instrumented model as shown in Section 3.1.
2. Adds to the model the class invariant as an axiom.
3. For each pair of object identifiers N_i, N_j, it performs the following analyses:

```
pred NiToNjInF[] {Ni+Nj in FReach and Ni->Nj in F}
run NiToNjInF for scopes
```

In the example, for field `fleft` we must check, for instance,

```
pred TNode0ToTNode1Infleft[ ] {
   TNode0+TNode1 in FReach and
   TNode0->TNode1 in fleft }
run TNode0ToTNode1Infleft for exactly 1 Tree,
   exactly 5 TNode, exactly 5 Data
```

If a "run" produces no instance, then there is no memory heap in which N_i`->`Nj `in F` that satisfies the class invariant. Therefore, the edge is infeasible within the provided scope. It is then removed from U_F, the upper bound relation associated to field `F` in the KodKod model. This produces tighter KodKod bounds which, when the KodKod model is translated to a propositional formula, yield a SAT problem involving fewer variables.

TACO's algorithm (whose pseudocode is given in Fig. 4), receives a collection of Alloy models to be analyzed, one for each edge whose feasibility must be checked. It also receives as input a threshold time T to be used as a time bound for the analyses. All the models are analyzed in parallel using the available resources. Those individual checks that exceed the time bound T are stopped and left for the next iteration. Each analysis that finishes as unsatisfiable tells us that an edge may be removed from the current bound. Satisfiable

```
global TIMEOUT

function fill_queue(upper_bounds, spec): int
  int task_count=0
  For each edge {f:A->B} in upper_bounds
    M := create_Alloy_model({f:A->B},upper_bounds,spec)
    task_count++
    ENQUEUE(<{f:A->B},M>, workQ)
  End For
  return task_count

function ITERATIVE_MASTER(scope, spec): upper_bounds
  workQ := CREATE_QUEUE()
  upper_bounds := initial_upper_bounds(spec, scope)
  Do
     task_count := fill_queue(upper_bounds, spec)
     result_count := 0,
     timeout_count := 0,
     unsat_count := 0;
     While result_count != tasks_count
       <{f:A->B}, analysis_result> := RECV()
       result_count++
       If analysis_result==UNSAT then
         upper_bounds := upper_bounds - {f:A->B}
       else If analysis_result==TIMEOUT then
         timeout_count ++
     End While
     if unsat_count==0
       return upper_bounds
  Until timeout_count==0
  return upper_bounds

function ITERATIVE_SLAVE()
  While workQ>0
    <{f:A->B},M> := DEQUEUE()
    analysis_result := run_stoppable_Alloy(M, TIMEOUT)
    SEND(master, <{f:A->B}, analysis_result>)
  End While
```

Figure 4: TACO's algorithm for bound refinement.

checks tell us that the edge cannot be removed. After all the models have been analyzed, we are left with a partition of the current set of edge models in three sets: unsatisfiable checks, satisfied checks, and stopped checks for which we do not have a conclusive answer. We then refine the bounds (using the information from the unsatisfiable models) for the models whose checks were stopped. The formerly stopped models are sent again for analysis. This leads to an iterative process that, after a number of iterations, converges to a (possibly empty) set of models that cannot be checked (even using the refined bounds) within the threshold T. Then, the bounds refinement process finishes. Notice that, in TACO's algorithm, the most complex analyses (those reaching the timeout) get to use tighter bounds in each iteration.

The following theorem shows that the bound refinement process is safe, i.e., it does not miss bugs.

THEOREM 3.3. *Let H be a memory heap exposing a bug. Then there exists a memory heap H' exposing the bug that satisfies the instrumentation and such that for each field* `g`*, the set of edges with label* `g` *(or* `bg` *or* `fg` *in case* `g` *is recursive) is contained in the refined $U_{\mathtt{g}}$.*

For all the case studies we are reporting in Section 4 it was possible to check all edges using this algorithm. Since bounds only depend on the class invariant, the signatures scope and the typing of the method under analysis, the same bound can be used (as will be seen in Section 4) to improve the analysis of different methods. Therefore, once a bound has been computed, it is stored in a bounds repository, as shown in Fig. 1.

4. EXPERIMENTAL RESULTS

In this section we report two kinds of experiments. We first analyze 6 collection classes with increasingly complex class invariants. Using these classes we will compare the performance of TACO against the performance of TACO$^-$. TACO$^-$ implements the same translation to a SAT problem implemented in TACO, but does not instrument the intermediate Alloy model using symmetry reduction axioms, and does not use tight bounds. We also compare with JForge [10] (a state-of-the-art SAT-based analysis tool developed at MIT). Since these classes are considered bug-free, this allows us to compare the tools in a situation where the state space must be exhausted. Later in the section, we compare the bug-finding capabilities of TACO against several state-of-the-art tools based on SAT-solving, model checking and SMT-solving.

4.1 Experimental Setup

The parallel algorithm for computing bounds was run in a cluster of 16 identical quad-core PCs (64 cores total), each featuring two Intel Dual Core Xeon processors running at 2.67 GHz, with 2 MB (per core) of L2 cache and 2 GB (per machine) of main memory. Non-parallel analyses, such as those performed with TACO *after* the bounds were computed, or when using other tools, were run on a single node. The cluster OS was Debian's "etch" flavor of GNU/Linux (kernel 2.6.18-6). The message-passing middleware was version 1.1.1 of MPICH2, Argonne National Laboratory's portable, open-source implementation of the MPI-2 Standard. All times are reported in `mm:ss` format.

4.2 Analysis of Bug-Free Code

In this section we analyze methods from collection classes with increasingly rich invariants. We will consider the following classes:

LList: An implementation of sequences based on singly linked lists.

AList: The implementation `AbstractLinkedList` of interface `List` from the Apache package `commons.collections`, based on circular doubly-linked lists.

CList: The implementation `NodeCachingLinkedList` of interface `List` from the Apache package `commons.collections`.

TreeSet: The implementation of class `TreeSet` from package `java.util`, based on red-black trees.

AVLTree: An implementation of AVL trees obtained from the case study used in [2].

BHeap: An implementation of binomial heaps used as part of a benchmark in [30].

In Section 3 we emphasized the fact that our technique allowed us to remove variables in the translation to a propositional formula. Each of the reported classes includes some field definitions. For each field `f` in a given class, during the translation from Alloy to KodKod an upper bound $U_{\mathtt{f}}$ is readily built. We will call the union of the upper bounds over all fields, the *upper bound*. In Table 1 we report, for each class, the following:

1. The number of variables used by TACO$^-$ in the upper bound (#UB). That is, the size of the upper bound without using the techniques described in this article.

2. The size of the tight upper bound (#TUB) used by TACO. The tight upper bound is obtained by applying the bound refinement algorithm from Section 3.2 starting from the upper bound.

3. The time required by the algorithm in Fig. 4 to build the tight upper bound (the time required to build the initial upper bound is negligible).

In all cases, the timeout used during bound refinement for the individual analyses was set to 2'. Notice that, in average, over 70% of the variables in the bounds can be removed.

#Node		5	7	10	12	15	17
LList	#UB	30	56	110	156	240	306
	#TUB	9	13	19	23	29	33
	Time	00:11	00:14	00:23	00:36	01:01	01:23
AList	#UB	76	128	252	344	512	676
	#TUB	33	47	68	82	103	117
	Time	00:16	00:25	00:51	01:26	02:47	09:28
CList	#UB	328	384	498	594	768	904
	#TUB	97	127	172	210	240	277
	Time	00:57	01:13	01:45	02:25	05:27	21:31
TrSet	#UB	170	280	650	852	1200	2006
	#TUB	59	107	200	279	424	533
	Time	00:49	01:13	03:03	05:11	11:30	44:23
AVL	#UB	150	280	650	852	1200	2006
	#TUB	55	98	177	251	389	491
	Time	00:33	00:57	03:26	09:53	22:03	101:31
BHeap	#UB	222	360	803	1053	1488	2394
	#TUB	75	123	218	293	423	481
	Time	00:44	01:12	04:00	06:48	20:13	62:50

Table 1: Sizes for initial upper bounds (#UB) and for tight upper bounds (#TUB), and analysis time for computation of tight upper bounds.

Once we have computed the bounds, we compare the analysis times for methods in the studied classes using TACO$^-$, JForge and TACO, as seen in Table 2. In all cases we are checking that the invariants are preserved. Also, for classes `LList`, `AList` and `CList`, we show that methods indeed implement the sequence operations. Similarly, in class `TreeSet` we also show that methods correctly implement the corresponding set operations. For class `BHeap` we also show that methods correctly implement the corresponding priority queue operations. Loops are unrolled up to 10 times, and no contracts for called methods are being used (just their inlined code). In each column we consider different scopes for the nodes signature. We set the scope for signature `Data` equal to the scope for nodes. We have set a timeout (TO) of 10 hours for each one of the analyses. Entries "OofM" mean "out of memory error". When reporting times using TACO, we are not adding the times (given in Table 1) to compute the bounds. Still, adding these times does not yield a TO for any of the analyses that did not exceed 10 hours. The code being analyzed was supposedly bug-free. Actually, when analyzing method `extractMin` in class BHeap, an error was detected using TACO. Boldface positions in the table signal that the analysis time reported is the one when the bug was found.

Looking in Table 2 at the progression of analysis times for TACO without bounds and JForge, it is clear that TACO with tight bounds requires in most cases several orders of magnitude less analysis time. While we will not present a detailed analysis of memory consumption, it is our experience that TACO uses less memory both during translation to a propositional formula and during SAT-solving than TACO$^-$ and JForge.

			5	7	10	12	15	17
LList	Contains	T$^-$	00:03	00:05	00:08	00:11	00:13	00:22
		JF	00:01	02:00	TO	TO	TO	TO
		T	00:03	00:04	00:05	00:06	00:07	00:09
	Insert	T$^-$	00:04	00:09	01:14	00:33	04:26	01:25
		JF	00:02	04:56	TO	TO	TO	TO
		T	00:04	00:05	00:07	00:08	00:13	00:26
	Remove	T$^-$	00:05	00:27	TO	TO	TO	TO
		JF	00:04	21:51	TO	TO	TO	TO
		T	00:04	00:06	00:11	00:12	00:17	00:33
AList	Contains	T$^-$	00:05	00:11	00:29	00:38	00:42	01:20
		JF	00:02	05:01	TO	TO	TO	TO
		T	00:04	00:06	00:16	00:22	00:27	00:58
	Insert	T$^-$	00:04	00:05	01:02	26:22	TO	TO
		JF	00:03	11:52	TO	TO	TO	TO
		T	00:04	00:05	00:07	00:08	00:12	00:16
	Remove	T$^-$	00:06	00:14	11:25	347:39	TO	TO
		JF	00:18	73:27	TO	TO	TO	TO
		T	00:05	00:06	00:17	00:31	01:08	03:13
CList	Contains	T$^-$	00:46	03:51	00:22	01:01	01:30	06:39
		JF	00:05	10:23	TO	TO	TO	TO
		T	00:11	00:19	01:23	01:56	05:51	07:25
	Insert	T$^-$	00:11	22:22	TO	TO	TO	TO
		JF	00:20	201:54	TO	TO	TO	TO
		T	00:09	00:12	00:16	00:28	01:07	02:01
	Remove	T$^-$	02:43	TO	TO	TO	TO	TO
		JF	02:28	TO	TO	TO	TO	TO
		T	00:27	00:59	03:26	03:43	28:18	57:23
TreeSet	Find	T$^-$	02:13	276:49	TO	TO	TO	TO
		JF	00:42	117:49	TO	TO	TO	TO
		T	00:04	00:10	01:56	12:43	58:54	305:06
	Insert	T$^-$	21:38	TO	TO	TO	TO	TO
		JF	OofM	OofM	OofM	OofM	OofM	OofM
		T	00:43	08:44	TO	TO	TO	TO
AVL	Find	T$^-$	00:14	27:06	TO	TO	TO	TO
		JF	00:26	190:10	TO	TO	TO	TO
		T	00:03	00:06	00:36	01:41	08:20	33:06
	FindMax	T$^-$	00:02	00:04	46:12	TO	TO	TO
		JF	00:06	49:49	TO	TO	TO	TO
		T	00:01	00:01	00:03	00:04	00:09	00:13
	Insert	T$^-$	01:20	335:51	TO	TO	TO	TO
		JF	OofM	OofM	OofM	OofM	OofM	OofM
		T	00:07	00:34	04:47	21:53	173:57	TO
BHeap	FindMin	T$^-$	00:12	11:41	TO	TO	TO	TO
		JF	00:22	83:07	TO	TO	TO	TO
		T	00:05	00:08	00:14	00:17	01:31	02:51
	Decrease Key	T$^-$	05:36	TO	TO	TO	TO	TO
		JF	01:48	TO	TO	TO	TO	TO
		T	00:16	01:13	30:26	TO	TO	TO
	Insert	T$^-$	22:46	391:10	TO	TO	TO	TO
		JF	73:47	TO	TO	TO	TO	TO
		T	01:54	08:08	37:30	218:13	TO	TO
	Extract Min	T$^-$	99:59	TO	TO	TO	TO	TO
		JF	73:47	OofM	OofM	OofM	OofM	TO
		T	01:13	14:01	36:52	TO	**43:33**	**176:47**

Table 2: Comparison of code analysis times for 10 loop unrolls using TACO$^-$(T$^-$), JForge (JF) and TACO (T).

4.3 Bug Detection Using TACO

In this section we report on our experiments using TACO in order to detect errors, and will compare TACO to other tools. We will be analyzing method `Remove` from classes `LList` and `CList`, and method `ExtractMin` from class `BHeap`.

Due to the similarities in the analysis techniques, we will first compare TACO with TACO$^-$ and JForge, and later in the section we will also compare with ESC/Java2 [5], JavaPathFinder [29], and Sireum/Kiasan [8]. We also used Jahob [3], which neither succeeded in verifying the provided specifications, nor provided an understandable counterexample (only raw formulas coming from the SMT-solvers).

In order to compare JForge, TACO$^-$ and TACO we will generate mutants for the chosen methods using the muJava [22] mutant generator tool. After manually removing from the mutants set those mutants that either were equivalent to the original methods or only admitted infinite behaviors (the latter cannot be killed using these tools), we were left with 31 mutants for method `Remove` from class `LList`, 81 mutants for method `Remove` from class `CList` and 50 mutants for method `ExtractMin` from class `BHeap`.

For all the examples in this section we have set the analysis timeout in 1 hour.

In Fig. 5 we report, for each method, the percentage of mutants that can be killed as the scope for the `Node` signature increases. We have set the scope for signature `Data` equal to the number of nodes. Notice that while the 3 tools behave well in class `LList`, TACO can kill strictly more mu-

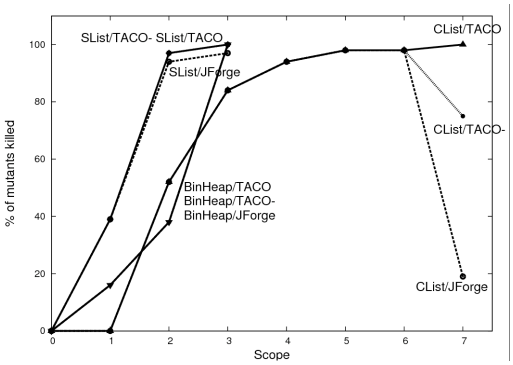


Figure 5: Efficacy of JForge, TACO$^-$ and TACO for mutants killing.

tants than TACO$^-$ and JForge the CList example. We can also see that as the scope increases, TACO$^-$ and JForge can kill fewer mutants. This is because some mutants that were killed in smaller scopes cannot be killed within 1 hour in a larger scope.

In order to report analysis times, we will carry out the following procedure, which we consider the most appropriate for these tools:

1. Try to kill each mutant using scope 1. Let T_1 be the sum of the analysis times using scope 1 for all mutants. Some mutants will be killed, while others will survive. For the latter, the analysis will either return UNSAT (no bug was found in that scope), or the 1 hour analysis timeout will be reached.

2. Take the mutants that survived in step 1, and try to kill them using scope 2. Let T_2 be the sum of the analysis times.

3. Since we know the minimum scope k for which all mutants can be killed (because TACO reached a 100% killing rate without any timeouts in scope k), repeat the process in step 2 until scope k is reached. Finally, let $T = \sum_{1 \leq i \leq k} T_i$.

Notice first that the previous procedure favors TACO$^-$ and JForge. In effect, if a tool is used in isolation we cannot set an accurate scope limit beforehand (it is the user's responsibility to set the limit). If a scope smaller than the necessary one is chosen, then killable mutants will survive. If a scope larger than the appropriate one is set, then we will be adding 1 hour timeouts that will impact negatively on the reported times. Notice also that an analysis that reached the timeout for scope $i < k$ will be run again in scope $i+1$. This is because we cannot anticipate if the timeout was due to a performance problem (the bug can be found using scope i but the tool failed to find the bug within 1 hour), or because the bug cannot be found using scope i. In the latter case it may happen that the mutant can be found in scope $i+1$ before reaching the timeout. This is the situation in Table 2 for method `ExtractMin` where the timeout was reached by TACO for scope 12, yet the bug was found using scope 15.

It is essential to notice that the same tight bound is used by TACO for killing all the mutants for a method within a given scope. Thus, when reporting analysis times for TACO in Table 3, we also add the time required to compute the bounds for scopes $1, \ldots, k$. In general we tried to use 10 loop unrolls in all cases. Unfortunately, JForge runs out of memory for more than 3 loop unrolls in the ExtractMin experiment. Therefore, for this experiment, we are considering only 3 loop unrolls for JForge, TACO$^-$ and TACO.

	JForge	TACO$^-$	TACO
LList.Remove	01:49	06:56	08:36 + 00:40
CList.Remove	891:50	245:12	34:51 + 06:35
BHeap.ExtractMin	04:34	19:35	16:06 + 01:09

Table 3: Analysis times for mutants killing. TACO times reflect the analysis time plus the bounds computation time.

In order to compare with tools based on model checking and SMT-solving, we will carry out the following experiments. We will choose the most complex mutants for each method. For class `LList` we chose mutant AOIU_1, the only mutant of method `Remove` that cannot be killed using scope 2 (it requires scope 3). For class `CList` we chose mutants AOIS_31 and AOIS_37, the only ones that require scope 7 to be killed. Finally, for class `BHeap` there are 31 mutants that require scope 3 to be killed (all the others can be killed in scope 2). These can be grouped into 7 classes, according to the mutation operator that was applied. We chose one member from each class. In Table 4 we present analysis times using all the tools. Table 4 shows that TACO, Java PathFinder and Kiasan were the only tools that succeeded in killing all the mutants. Since the fragment of JML supperted by ESC/Java2 is not expressive enough to model the invariant from class BHeap, we did not run that experiment.

	JForge	TACO$^-$	ESCJ	Kiasan	JPF	TACO
LList.AOIU_1	00:01	00:09	00:06	00:05	00:02	00:18
CList.AOIS_31	TO	TO	TO	00:13	02:55	01:00
CList.AOIS_37	TO	TO	TO	00:14	02:18	01:02
BHeap.AOIS_41	00:08	00:13	–	00:32	00:03	00:13
BHeap.AOIU_8	00:02	00:14	–	00:26	00:04	00:12
BHeap.AORB_10	00:04	00:14	–	00:26	00:24	00:12
BHeap.COI_22	00:01	00:11	–	01:05	00:03	00:10
BHeap.COR_5	00:01	00:08	–	00:15	00:25	00:10
BHeap.LOI_15	00:02	00:11	–	00:26	00:29	00:15
BHeap.ROR_23	00:01	00:11	–	00:16	00:04	00:09

Table 4: Comparison of analysis behavior for some selected mutants. Analysis time for TACO includes the time required to compute the tight bound amortized among the mutants in each class.

Notice that although we chose the supposedly most complex mutants, these are still simple in the sense that they can be killed using small scopes. As mentioned in Section 4.2, TACO found a previously unreported bug in method `ExtractMin` from class `BHeap`. The bug cannot be reproduced using mutation because the smallest input that produces a failure has 13 nodes (and all mutants can be killed with just 3 nodes). The input datum leading to the failure is presented in Fig. 6. Notice that at least 4 loop unrolls were required in TACO in order to exhibit the failure. In Table 5 we report analysis times when attempting to discover the bug using all the tools. TACO is the only tool that succeeded in discovering the error. The analysis time for TACO reports the time for computing the bound, plus the analysis time using 4 loop unrolls.

	JForge	TACO$^-$	Kiasan	JPF	TACO
BHeap. ExtractMin	TO	TO	OofM	TO	20:13 + 00:53

Table 5: Analysis of a non-trivial bug.

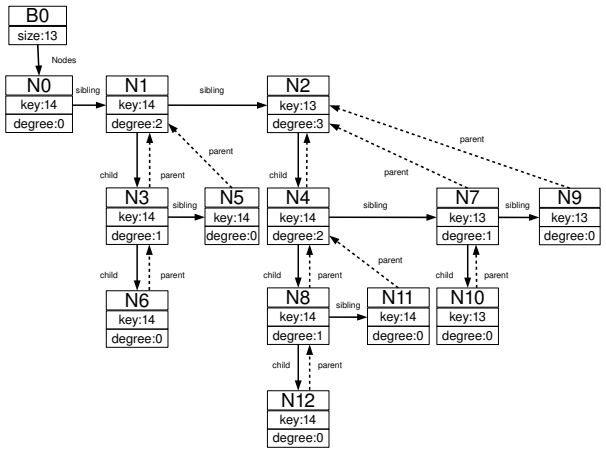


Figure 6: A 13 nodes heap that exhibits the failure in method `ExtractMin`.

4.4 Threats to Validity

We begin by discussing how representative the selected case studies are. As discussed in [30], container classes have become ubiquitous. Therefore, providing confidence about their correctness is an important task in itself. But, as argued in [26], these structures (which combine list-like and tree-like structures) are representatives of a wider class of structures including, for instance, XML documents, parse trees, etc. Moreover, these structures have become accepted benchmarks for comparison of analysis tools in the ISSTA community (see for instance [4, 9, 19, 30]).

In all experiments we are considering the performance of TACO$^-$ as a control variable that allows us to guarantee that TACO's performance improvement is due to the presented techniques.

In Section 4.2 we analyzed bug-free code. Since the process of bug finding ends when no more bugs are found, this situation where bug free code is analyzed is not artificial, but is rather a stress test that necessarily arises during actual bug finding.

In Section 4.3 we have compared several tools. It is not realistic to claim that every tool has been used to the best of its possibilities. Yet we have made our best efforts in this direction. In the case of JForge, since it is very close to TACO, we are certain we have made a fair comparison. For Java PathFinder and Kiasan we were careful to write repOK invariant methods in a way that would return false as soon as an invariant violation could be detected. For ESC/Java2, since it does not support any constructs to express reachability, we used weaker specifications that would still allow the identification of bugs. For Jahob we used Jahob's integrated proof language, and received assistance from Karen Zee in order to write the models. More tools could have been compared in this section. Miniatur and FSoft are not available for download even for academic use, and therefore were not used in the comparison. Other tools such as CBMC and Saturn (designed for analysis of C code) departed too much from our intention to compare tools for the analysis of Java code.

Analysis using TACO requires using a cluster of computers to compute tight bounds. Is it fair to compare with tools that run on a single computer? While we do not have a conclusive answer, for the bug in method `ExtractMin` (even considering the time required to compute the bounds *sequentially*) TACO seems to outperform the sequential tools. This is especially clear in those cases where the sequential tools run out of memory before finding the bug (as is the case for Kiasan and JForge). More experiments are required in order to provide a conclusive answer.

5. RELATED WORK

In Section 3.1 we analyzed related work on heap canonicalization. In Section 4 we compared our tool with several other state-of-the-art tools for program analysis. In this section we review related (but difficult to compare experimentally) work.

The Alloy Annotation Language (AAL) was introduced in [20]. It allows the annotation of Java-like code using Alloy as the annotation language. The translation proposed in [20] does not differ in major ways from the one we implement. Analysis using AAL does not include any computation of bounds for fields.

In [28] the authors present a set of rules to be applied along the translation to a SAT-formula in order to profit from properties of functional relations. The article presents a case-study where insertion in a red-black tree is analyzed. The part of the red-black tree invariant that constrains trees to not have two consecutive red nodes is shown to be preserved. In our experiment we verify that the complete (significantly more complex) invariant is preserved. Actually, for 8 loop unrolls and scope 7 for nodes and data, the analysis time decreases from 08:53 (for the property we analyze) to 0.153 seconds using the weakened property.

Saturn [32] is also a SAT-based static analysis tool for C. It uses as its main techniques a slicing algorithm and function summaries. As in our case, sequential code is faithfully modeled at the intraprocedural level (no abstractions are used). Unlike TACO, summaries of called functions may produce spurious counterexamples. Saturn can check assertions written as C "assert" statements. Its assertion language is not as declarative as our extension of JML.

F-Soft [17] also analyses C code. It computes ranges for values of integer valued variables and for pointers under the hypothesis that runs have bounded length. It is based on the framework presented in [24]. Our technique produces tighter upper bounds because it does not compute feasible intervals for variables, but instead checks each individual value.

Unlike techniques based on abstraction that require the user to provide *core properties* (i.e., *shape analysis* [25]), TACO does not require user-provided properties other than the JML annotations.

The techniques presented in the article are quite general. Explicit state model checkers (such as JPF) can use tight bounds in order to prune the state space when a state contains edges that lay outside the bound. Korat [4] can avoid

evaluating the `repOk` method whenever the state is not contained in the bounds. Running a simple membership test will many times be less expensive than running a `repOk` method. Tools that are similar to TACO (such as Miniatur and JForge), can make direct use of these techniques.

6. CONCLUSIONS AND FURTHER WORK

This article shows that a methodology based on (1) adding appropriate constraints to SAT problems, and (2) using the constraints to remove unnecessary variables, makes SAT-solving a method for program analysis as effective as model checking or SMT-solving.

The experimental results presented in the article show that bounds can be computed effectively, and that once bounds have been computed, the analysis time improves considerably. This allowed us to analyze real code using domain scopes beyond the capabilities of current similar techniques and find bugs that cannot be detected using state-of-the-art tools for bug-finding. Still, this article presents a naive approach to bound computation, and is a starting point to a new research line on *efficient* bound computation.

7. ACKNOWLEDGEMENTS

We thank Elena Morin for proofreading this article.

8. REFERENCES

[1] Andoni, A., Daniliuc, D., Khurshid, S. and Marinov, D., *Evaluating the "Small Scope Hypothesis"*, downloadable from `http://sdg.csail.mit.edu/publications.html`.

[2] Belt, J., Robby and Deng X., *Sireum/Topi LDP: A Lightweight Semi-Decision Procedure for Optimizing Symbolic Execution-based Analyses*, FSE 2009, pp. 355–364.

[3] Bouillaguet Ch., Kuncak V., Wies T., Zee K., Rinard M.C., *Using First-Order Theorem Provers in the Jahob Data Structure Verification System*. VMCAI 2007, pp. 74–88.

[4] Boyapati C., Khurshid S., Marinov D., *Korat: automated testing based on Java predicates*, in ISSTA 2002, pp. 123–133.

[5] Chalin P., Kiniry J.R., Leavens G.T., Poll E. *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*. FMCO 2005: 342-363.

[6] Clarke E., Kroening D., Lerda F., *A Tool for Checking ANSI-C Programs*, in TACAS 2004, LNCS 2988, pp. 168–176.

[7] deMillo R. A., Lipton R. J., Sayward F. G., *Hints on Test Data Selection: Help for the Practicing Programmer*, in IEEE Computer pp. 34–41, April 1978.

[8] Deng, X., Robby, Hatcliff, J., *Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs*, in SEFM 2007, pp. 273-282.

[9] Dennis, G., Chang, F., Jackson, D., *Modular Verification of Code with SAT*. in ISSTA'06, pp. 109–120, 2006.

[10] Dennis, G., Yessenov, K., Jackson D., *Bounded Verification of Voting Software*. in VSTTE 2008. Toronto, Canada, October 2008.

[11] Dolby J., Vaziri M., Tip F., *Finding Bugs Efficiently with a SAT Solver*, in ESEC/FSE'07, pp. 195–204, ACM Press, 2007.

[12] Flanagan, C., Leino, R., Lillibridge, M., Nelson, G., Saxe, J., Stata, R., *Extended static checking for Java*, In PLDI 2002, pp. 234–245.

[13] Frias, M. F., Galeotti, J. P., Lopez Pombo, C. G., Aguirre, N., *DynAlloy: Upgrading Alloy with Actions*, in ICSE'05, pp. 442–450, 2005.

[14] Frias, M. F., Lopez Pombo, C. G., Galeotti, J. P., Aguirre, N., *Efficient Analysis of DynAlloy Specifications*, in ACM-TOSEM, Vol. 17(1), 2007.

[15] Galeotti, J. P., Frias, M. F., *DynAlloy as a Formal Method for the Analysis of Java Programs*, in Proceedings of IFIP Working Conference on Software Engineering Techniques, Warsaw, 2006, Springer.

[16] Iosif R., *Symmetry Reduction Criteria for Software Model Checking*. SPIN 2002: 22-41

[17] Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P., *F-Soft: Software Verification Platform*. In CAV'05, pp. 301–306, 2005.

[18] Jackson, D., *Software Abstractions*. MIT Press, 2006.

[19] Jackson, D., Vaziri, M., *Finding bugs with a constraint solver*, in ISSTA'00, pp. 14-25, 2000.

[20] Khurshid, S., Marinov, D., Jackson, D., *An analyzable annotation language*. In OOPSLA 2002, pp. 231-245.

[21] Khurshid, S., Marinov, D., Shlyakhter, I., Jackson, D., *A Case for Efficient Solution Enumeration*, in SAT 2003, LNCS 2919, pp. 272–286.

[22] Ma Y-S., Offutt J. and Kwon Y-R., *MuJava : An Automated Class Mutation System*, Journal of Software Testing, Verification and Reliability, 15(2):97-133, 2005.

[23] Musuvathi M., Dill, D. L., *An Incremental Heap Canonicalization Algorithm*, in SPIN 2005: 28-42

[24] Rugina, R., Rinard, M. C., *Symbolic bounds analysis of pointers, array indices, and accessed memory regions*, in PLDI 2000, pp. 182–195, 2000.

[25] S. Sagiv, T. W. Reps, R. Wilhelm. *Parametric shape analysis via 3-valued logic*. ACM TOPLAS 24(3): 217–298 (2002)

[26] Siddiqui, J. H., Khurshid, S., *An Empirical Study of Structural Constraint Solving Techniques*, in ICFEM 2009, LNCS 5885, 88–106, 2009.

[27] Torlak E., Jackson, D., *Kodkod: A Relational Model Finder*. in TACAS '07, LNCS 4425, pp. 632–647.

[28] Vaziri, M., Jackson, D., *Checking Properties of Heap-Manipulating Procedures with a Constraint Solver*, in TACAS 2003, pp. 505-520.

[29] Visser W., Havelund K., Brat G., Park S. and Lerda F., *Model Checking Programs*, ASE Journal, Vol.10, N.2, 2003.

[30] Visser W., Păsăreanu C. S., Pelánek R., *Test Input Generation for Java Containers using State Matching*, in ISSTA 2006, pp. 37–48, 2006.

[31] Visser W., *Private communication*, February 2nd., 2010.

[32] Xie, Y., Aiken, A., *Saturn: A scalable framework for error detection using Boolean satisfiability*. in ACM TOPLAS, 29(3): (2007).