TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds*

Juan P. Galeotti, Nicolás Rosner, Carlos G. López Pombo, and Marcelo F. Frias

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 failure 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 which, on one hand, reduces the size of the search space by ignoring certain classes of isomorphic models, and on the other hand, 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 to an improvement of 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.

Index Terms—Static analysis, SAT-based code analysis, Alloy, KodKod, DynAlloy.

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 [10], Saturn [47] and F-Soft [28] for the analysis of C code, and by Miniatur [20] and JForge [15] 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 [15]. 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.

SAT-based analysis of code has been perceived as an intrinsically non-scalable technique. The reason is that the translation of a complete system to a propositional formula, and the analysis of such a formula using a SAT-solver, are very likely not to scale. We believe this is mostly true unless some careful decisions are made. For instance, it is worth accepting that SAT-based analysis (as described) is not meant to be a monolithic process to be applied to large pieces of software. Also, it is important to understand the reasons for the non-scalability of SAT-solving, and act in order to minimize their impact during analysis. Finally, it is essential to fully understand what are the benefits of SAT-based analysis when compared to other analysis techniques.

The contribution of this article is twofold. From the methodological point of view we make a case for a responsible adoption of SAT-based analysis of code. From the technical point of view, we present a novel, general and fully automated technique for the intra-procedural analysis of JML-annotated Java code, in a way consistent with the methodology hereby presented. Both the methodology and the technique presented in this article are supported by our prototype tool, TACO (Translation of Annotated COde).

It is well known that the SAT problem is NP-complete [11]. Thus, the time required for solving an instance of this problem is (provided $P \neq NP$) exponential on the amount of propositional variables of the formula resulting from the translation of source code. In order to improve the analysis time we can then proceed in two ways:

- Reducing the number of propositional variables in

This article is a revised and extended version of [26].

- *Juan P. Galeotti is with Saarland University – Computer Science Saarbrücken, Germany. E-mail: jgaleotti@cs.uni-saarland.de*
- *Carlos G. López Pombo is with Department of Computer Science, FCEyN, Universidad de Buenos Aires, and CONICET, Argentina. E-mail: clpombo@dc.uba.ar*
- *Nicolás Rosner is with Department of Computer Science, FCEyN, Universidad de Buenos Aires, Argentina. E-mail: nrosner@dc.uba.ar*
- *Marcelo F. Frias is with Instituto Tecnológico de Buenos Aires (ITBA), and CONICET, Argentina. E-mail: mfrias@itba.edu.ar*

- *This research has been funded by MINCyT under grant PICT 2010-1690.*

the propositional formula result of the translation.
- Reducing the number of valuations to be considered by the SAT-solver by removing valuations whose analysis we know in advance will not lead to a fault.

The TACO technique actually combines both approaches in a synergic and fully automated way. First, it forces a canonical representation of the Java memory heap by removing permutations (also called *symmetries*) of object references. This greatly reduces the number of meaningful valuations of the initial state to be considered by the SAT-solver. Second, and as a consequence of the heap canonicalization, a simple preprocessing makes it possible to determine in advance the truth value of a substantial proportion of the propositional variables. These variables can be replaced by their predetermined truth value, yielding a simpler SAT problem.

As a hint of the power of these techniques, symmetry reduction by itself allows us to reduce the analysis time of a method for inserting an element in an AVL tree from over 10 hours to approximately 17 minutes. After removing propositional variables that TACO deemed unnecessary, the analysis time reduced to 5 minutes. The overall cluster computing time required by TACO to rule out those unnecessary propositional variables was of only 1 minute 55 seconds.

The contributions of this article 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 Sec. 3).
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 Sec. 4.1). 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 with complex data structures that show that the technique improves the analysis by reducing analysis times by several orders of magnitude in the cases where correct code is analyzed. We also show that the technique can efficiently discover faults seeded using mutant generation [12]. Finally, we report on a previously unknown [46] fault found in a benchmark presented in [45]. This fault was not detected by several state-of-the-art tools based on SAT-solving, model checking or SMT-solving.

The article is organized as follows. In Sec. 2 we describe the translation of JML-annotated sequential Java code to a SAT problem. In Secs. 3 and 4 we present the TACO technique for program analysis. In Sec. 5 we present the experimental results. In Sec. 6 we discuss related work. Finally, in Sec. 7 we discuss lines for further work and draw conclusions about the results presented in the article.

2 Translating JML to Alloy

In this section we present an outline of our translation of JML [22] annotated Java code to a SAT problem. In intention, the translation is not very different from translations previously presented by other authors [19] or by some of the authors of this article [25]. 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 in this article we use 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`.

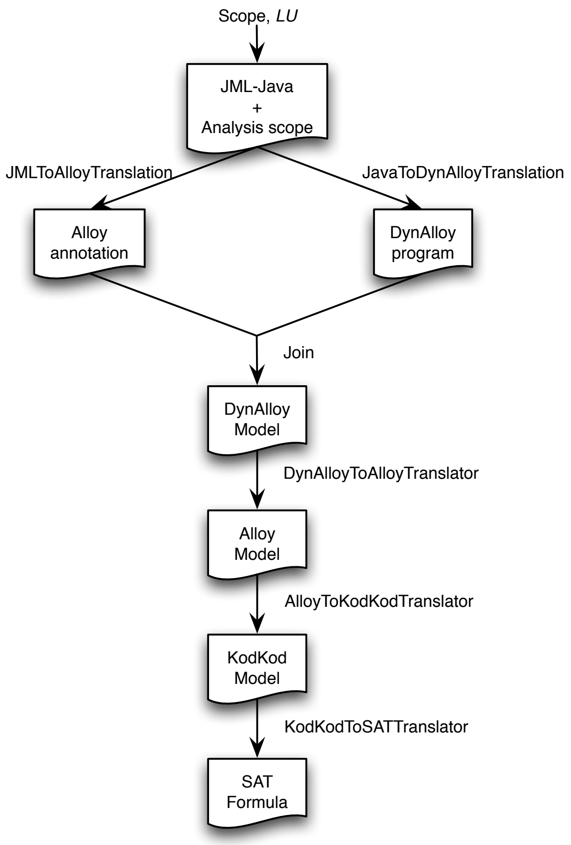


Fig. 1. Translating annotated code to SAT.

Our translation uses Alloy [29] as an intermediate language. This is an appropriate decision because Alloy is close to JML, and the Alloy Analyzer [29] provides a simple interface to several SAT-solvers. Also, Java code can be translated to DynAlloy programs [25]. DynAlloy [23] is an extension of Alloy that allows us to specify actions that modify the state in much the same way 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 that model sequential Java code.

DynAlloy is based on first-order dynamic logic [36]. The aim of this specification language is to provide a formal characterization of imperative sequential programs. Fig. 2 shows a relevant fragment of DynAlloy's grammar. It is worth noticing that more complex programming structures can be described using these basic logical constructs. For example, `if` B `then` P `else` Q `fi` can be written as the following DynAlloy program $B?; P + (\neg B)?; Q$. Similarly, `while` B `do` P `od` can be expressed as $(B?; P)^*; (\neg B)?$.

$$
\begin{array}{lll}
formula & ::= & \ldots \mid \{formula\}\ program\ \{formula\} \\
 & & \quad \text{“partial correctness”} \\
 & & \\
program & ::= & \langle formula, formula\rangle(\overline{x}) \\
 & & \quad \text{“atomic action”} \\
 & \mid & formula? \\
 & & \quad \text{“test”} \\
 & \mid & program + program \\
 & & \quad \text{“non-deterministic choice”} \\
 & \mid & program; program \\
 & & \quad \text{“sequential composition”} \\
 & \mid & program^* \\
 & & \quad \text{“iteration”} \\
 & \mid & \langle program\rangle(\overline{x}) \\
 & & \quad \text{“invoke program”}
\end{array}
$$

Fig. 2. DynAlloy grammar

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. A contract may include normal behavior (how the system behaves 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 linking nodes of type LNode containing 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. Failures could be detected by repeating the analysis using larger scopes; if an analysis does not find a failure, it means no failure exists *within* the provided scope for data domains. Therefore, only a portion of the program domain is actually analyzed. Fortunately, using bounded scopes is sufficient to expose many failures, since they can often be reproduced with few data [1].

The annotations are then translated to Alloy formulas using translation `JMLtoAlloyTranslation` [25], and the method under analysis is translated to a DynAlloy program using translation `JavaToDynAlloyTranslation` [25]. 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[1]) analysis will only expose failures 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 [20] and JForge [15].

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 extensively discussed in [24], 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 Sec. 4 fits in the code analysis process.

To describe the translation at a high level of abstraction let us consider the following Java classes for implementing singly-linked structures:

```
public class List {
  LNode head;
}

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

For the above Java classes, the resulting Alloy model includes the signature definitions shown below:

```
one sig null {}

sig List {
  head : LNode + null
}

sig LNode {
  next : LNode + null,
  key : Integer + null
}

sig Integer {}
```

According to Alloy's semantics, signatures define sets of atoms. The modifier `one` in signature `null` constrains

1. The use of the term *static* refers to the fact that code is not executed during analysis.

$$\begin{array}{c|cccccc}
M_{\mathtt{f}} & T_1 & T_2 & \dots & T_t & \mathtt{null} \\
\hline
S_1 & p_{S_1,T_1} & p_{S_1,T_2} & \cdots & p_{S_1,T_t} & p_{S_1,\mathtt{null}} \\
S_2 & p_{S_2,T_1} & p_{S_2,T_2} & \cdots & p_{S_2,T_t} & p_{S_2,\mathtt{null}} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
S_s & p_{S_s,T_1} & p_{S_s,T_2} & \cdots & p_{S_s,T_t} & p_{S_s,\mathtt{null}}
\end{array}$$

Fig. 3. Matrix representation of an Alloy field.

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)`.

The Alloy language has a relational semantics. This means that in order to translate an Alloy specification to a SAT problem, the technique focuses on the translation of fields as relations. 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 $S_1, \dots, S_s$ and $T_1, \dots, T_t$, respectively. Alloy uses a matrix $M_{\mathtt{f}}$ holding $s \times (t+1)$ propositional variables to represent the field `f` (see Fig. 3).

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 [42].

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 infrastructures to prototype the translation. Furthermore, we believe these languages better characterize the several semantic gaps when translating JML-annotated Java programs to a SAT problem.

3 A New Predicate for Symmetry Breaking

The process of SAT-based analysis relies on an implicit traversal of the space of plausible models (i.e., those that satisfy the specification) while looking for a model that does not satisfy the property being checked. As mentioned before, if this procedure finds one such model we know that a counterexample of the property exists. A model in this context is a valuation of the propositional variables. Thus, the size of the search space is exponential in the number of propositional variables, and we should strive to reduce its size.

Permutations of signature atoms (also called *symmetries*) do not alter the truth value of Alloy formulas. Therefore, once a valuation μ is considered, those valuations originated from μ by permuting atoms should be avoided. One way to do this is by introducing *symmetry breaking predicates* that rule out certain models. For instance, Alloy includes general-purpose symmetry breaking predicates [42].

In this section we present symmetry breaking predicates tailored to avoid permutations in the Alloy representation of the Java memory heap.

3.1 SAT-Based Symmetry Breaking

In order to describe predicates concisely we will use Alloy notation, which is thoroughly described in [29]. 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* (which 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 $^\wedge T$ 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_non_null[l : List] {
  all n : LNode |
    n in l.head.*next
      implies n !in n.^next and n.key!=null
}
```

describes acyclic lists that do not store *null* values. Running the predicate in the Alloy Analyzer using the command

```
run acyclic for exactly 1 List,
                exactly 4 LNode,
                exactly 1 Integer
```

yields (among others) the instances shown in Fig. 4. Notice that the list instance in the right-hand side is a permutation (on signature LNode) of the other one. This shows that while the symmetry breaking predicates

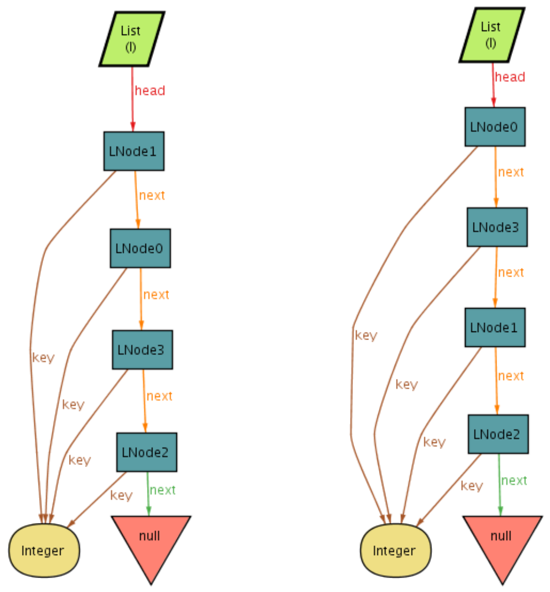


Fig. 4. Two isomorphic list instances found by Alloy Analyzer

included in Alloy remove many symmetries, some still remain. Actually, any permutation of LNode that stores data in the same order as any of these lists, is also a model. The ability to reduce the state space is central to scalability. Pruning the state space by removing permutations on signature LNode contributes to improving the analysis time by orders of magnitude.

Revisiting the singly linked lists example previously shown in Sec. 2, it is easy to see that a predicate forcing nodes to be traversed in the order LNode0 $\rightarrow$ LNode1 $\rightarrow$ LNode2 $\rightarrow \ldots$ removes all 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 [27], [38] present different ways of canonicalizing the heap ([27] uses a depth-first search traversal, while [38] 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 to a SAT-solver.

In the context of SAT-based analysis, [33] 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. JForge [14] reduces some symmetries by allocating fresh heap memory objects following a predefined total ordering of the atoms in the domain.

```
for each type $T$ do
   $k \leftarrow scope(T)$
   “one sig $T_1,\ldots,T_k$ extends $T$ {}”
end for

for each recursive $r : T \mapsto$ (one $T$ + null) do
   Add new field $fr : T \mapsto$ lone($T$ + null)
   Add new field $br : T \mapsto$ lone($T$)
   Replace each “$r$” usage with expression “$fr + br$”
   Remove field $r$
   Add new axiom:
   “fact {
      no ( $fr$.univ & $br$.univ ) and
      $T$ = $fr$.univ + $br$.univ
   }”
end for
```

Fig. 5. The *instrument_Alloy()* procedure

3.2 An Algorithm for Generating Symmetry Breaking Predicates

In this section we present a novel family of predicates that canonicalize arbitrary heaps.

Our model of Java memory 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 is the set of arcs, and contains pairs $\langle n_1, n_2 \rangle \in N \times N$. L is the arc labeling function. It assigns Java field names to arcs. An edge between nodes n_1 and n_2 labelled f_i means that $n_1.f_i = n_2$. The typing of fields must be respected. R is the root nodes labelling function, mapping the receiver variable *this*, method arguments and static class fields to nodes. For example, a node n labelled *this* means that in the heap representation, the receiver object is node n.

The algorithm depends on defining an enumeration function for types, fields and heap root elements. For the remainder of this section we will refer to $\{T_i\}_{i \in types}$, $\{f_i\}_{i \in fields}$ and $\{g_i\}_{i \in roots}$ as the ordered sets for types, fields and root nodes, respectively.

Instrumenting the Alloy Model

In order to include the predicates we will instrument the Alloy model obtained by the translation from the annotated source code.

Besides the sets of ordered types, fields and root nodes, it is required to provide the finite scope of analysis for each type in order to instantiate the axioms and their auxiliary functions.

Let us consider $scope(T)$, the function that returns for each type T the scope of analysis being used. The procedure *instrument_Alloy()* (shown in Fig. 5) starts by introducing a singleton atom denoting each element of type T within the scope of analysis.

Once the singletons have been introduced, the procedure continues by splitting every *recursive* field. A field is considered *recursive* if domain and codomain (minus the *null* value) match. For instance, field `next: LNode ↦ LNode+null` is considered a recursive field.

Each recursive field r from signature T is split into two *partial* functions (thus the `lone` modifier in Fig. 5): *fr* (the *forward* part of the field), mapping nodes to strictly greater nodes or `null`, and *br* (the *backward* part of the field), mapping nodes to lesser nodes. Non-recursive fields are not modified. As Java fields must be total functions, the procedure also adds new facts stating that for each recursive field r, the domains of *fr* and *br* form a partition of r's domain, making $fr + br$ a well–defined total function.

The new fields obtained (that substitute the original ones) are meant to split the set of the original edges between "forward" arcs and "backward" arcs. Forward arcs may only map nodes to *greater* nodes (in terms of the element index) or null, while backward arcs go to nodes that are *smaller or equal* in the ordering (and cannot go to null). Notice that forward arcs cannot lead to a cycle.

Because of the presented instrumentation, the set of original Alloy fields is partitioned into forward fields, backward fields, and non-recursive fields.

The instrumentation also modifies the facts, functions, predicates and assertions of the original model by replacing each occurrence of a recursive field r_i with the expression $fr_i + br_i$.

In the presence of subtypes, a transformation takes place before procedure *instrument_Alloy()* is executed. Subtypes are modeled using the atomization technique from [21]. Basically, an Alloy signature T does not represent the set of all objects whose Java static type is T, but only those objects of type T that do not belong to any subtype of T. The transformation decomposes each Alloy field into partial fields. Each new partial field maps atoms from a single Alloy signature to another (possibly equal) Alloy signature plus *null*. As with procedure *instrument_Alloy()*, this instrumentation also replaces each occurrence of a split field with the union of the associated partial fields. It also adds facts that enforce the union of the partial fields obtained from a split field to be a total function.

The Auxiliary Functions

The procedures shown in this subsection allow us to introduce the necessary auxiliary functions prior to introducing the symmetry breaking axioms.

Procedure *local_ordering()* (shown in Fig. 7) generates auxiliary functions for:

- establishing a linear order between elements of type T (function *next_T*).
- returning the least object (according to the ordering *next_T*) in an input subset (function *min_T*).
- returning the nodes in signature T smaller than the input parameter (function *prevs_T*).

Notice that all these functions are constrained to operations among the elements of type T. We will consider them as "local" ordering auxiliary functions.

On the other hand, procedure *global_ordering()* (shown in Fig. 8) is intended to provide functions which operate on all heap elements. This procedure defines Alloy functions for:

- establishing a linear order between elements of all types (function *globalNext*)
- returning the least object (according to the ordering *globalNext*) in an input subset (function *globalMin*).

Notice that function *globalNext* induces an ordering between types. In order to effectively remove all symmetries a sufficient condition on the ordering between types is the following:

> the ordering is such that whenever a heap may contain an object o of class T_1 pointing to an object o' of type T_2 (the latter being part of an heterogeneous cycle), type T_1 is less that type T_2.

In the previous paragraph, by *heterogeneous* we mean that the cycle must involve objects from at least two different classes. Such orderings are most times easy to find, and algorithm `type_ordering` (see Fig. 6) produces appropriate orderings for all the classes in the benchmark we will use in Sec. 5.

We will consider a node n' to be a parent of n if there exists a non-recursive field or a forward field f such that $n'.f = n$. A node may have no parents (in case it is a root node), or have several parent nodes. In the latter case, among the parents we will distinguish the minimal one (according to a global ordering) by calling it the *min-parent* of n. The procedure *define_min_parent()* (shown in Fig. 9) defines a min-parent function for each type T. If n belongs to type T, $minP_T[n]$ returns the min-parent of n (if any).

Notice that in the definition of function $minP_T$ we are only considering forward fields and non-recursive fields with target type T.

Key to the symmetry breaking predicates we are introducing is the notion of *reachable* objects. We consider a heap node to be *reachable* if it may be accessed during the program execution by traversing the memory heap.

Procedure *define_freach()* (presented in Fig. 10) defines a function `FReach` denoting all objects that may be reachable by accessing either non-recursive fields or forward fields. This definition is a more economic (regarding the translation to a propositional formula) description of the reachable heap objects since no mention to the backward fields is needed.

The Symmetry Breaking Predicates

The rest of the algorithm outputs axioms that canonicalize the order in which heap nodes are traversed. Intuitively, we will canonicalize heaps by ordering nodes according to their parents in the heap. We will explain the rest of the algorithm by considering the possibilities depicted in Fig. 11. Given two nodes of type T, we distinguish the following cases :

(a) Both nodes are root nodes.

```
**Procedure** type_ordering
Input: $IL$ = [ this, arg1, arg2, ... , argk ]
//Input list respects the parameter ordering
//of the method under analysis.
Output: list of types $OL$
//Output list $OL$ shows the type ordering.
$OL =$ emptyList
$F =$ list of fields in class under analysis
while nonEmpty($IL$) do
  $l = $ head($IL$)
  $t = $ type($l$)
  if $t$ !in $OL$ then
    $OL = OL$ concat BFS($t, F$)
  end if
  $IL = $ tail($IL$)
end while

Procedure BFS
Inputs:
  $t$ (type),
  $F =$ [field1, field2,..., fieldn]
//F is a list of fields from the class under analysis,
//in the order they were defined in the class.
Output: list of types $TL$
$TL = [t]$
$i = 0$
while $i < \mathrm{length}(TL)$ do
  for each field $f$ in $F$ do
    if type(domain($f$)) = $TL(i)$ then
      $t' = $ type(range($f$))
      if $t'$ !in $TL$ then
        $TL = TL$ concat $[t']$
      end if
    end if
  end for
  $i = i + 1$
end while
```

Fig. 6. An algorithm for type ordering.

```
for each type $T$ do
  $k \leftarrow scope(T)$
  "fun $nextT$[] : $T \rightarrow$ lone $T$ {
    $\langle T_1, T_2\rangle$ + $\langle T_2, T_3\rangle$ +...+ $\langle T_{k-1}, T_k\rangle$
  }
  fun $minT$[os: set $T$] : lone $T$ {
    os - os.^$nextT$[]
  }
  fun $prevsT$[o: $T$] : set $T$ {
    o.^($\sim nextT$[])
  }"
end for
```

Fig. 7. The *local_ordering()* procedure

```
"fun globalNext[] : Object $\rightarrow$ lone Object {"
for each type $T_i$ do
  if $i > 0$ then
    "+"
  end if
  "$\langle T_{i_1}, T_{i_2}\rangle$ + $\langle T_{i_2}, T_{i_3}\rangle$ +...+ $\langle T_{i_{k-1}}, T_{i_k}\rangle$"
  if $T_{i+1}$ exists then
    $max_{T_i} \leftarrow$ maximum singleton from $T_i$
    $min_{T_{i+1}} \leftarrow$ minimum singleton from $T_{i+1}$
    "+ $\langle max_{T_i}, min_{T_{i+1}}\rangle$ "
  end if
end for
"}
fun globalMin[s: set Object] : lone Object {
  s - s.^globalNext[]
}"
```

Fig. 8. The *global_ordering()* procedure

```
for each type $T$ do
  Let $f_1$,...,$f_i$ be the non-recursive fields targeting $T$
  Let $fr_1$,...,$fr_j$ be the forward fields targeting $T$
  Let $g_1$,...,$g_k$ be the root nodes of type $T$
  "fun $minP_T$[o : $T$] : Object {
    o !in ($g_1 + \cdots + g_k$)
      => globalMin[($f_1 + \ldots + f_i + fr_1 + \ldots + fr_j$).o]
      else none }"
end for
```

Fig. 9. The *define_min_parent()* procedure

(b) One node is a root node and the other is a non-root node.
(c) Both nodes are non-root nodes with the same min-parent.
(d) Both nodes are non-root nodes with different min-parents of the same type T'.
(e) Both nodes are non-root nodes with min-parents of different types.

Notice that any pair of nodes of type T is included in one (and only one) of these cases.

Procedure *order_root_nodes()* (presented in Fig. 12) outputs an axiom that sorts two root nodes of type T. The axiom forces every pair of root nodes to obey the ordering in which formal parameters and static fields (namely, the root nodes) were declared in the source Java file.

Procedure *root_is_minimum()* (presented in Fig. 13) creates an axiom that constrains the first non-null root

```
Let $f_1$,...,$f_i$ be the non-recursive fields
Let $fr_1$,...,$fr_j$ be the forward fields
Let $g_1$,...,$g_k$ be the root nodes
"fun FReach[] : set Object {
  ($g_1$+...+$g_k$).*($f_1$+...+$f_k$+$fr_1$+...+$fr_n$) - null
}"
```

Fig. 10. The *define_freach()* procedure

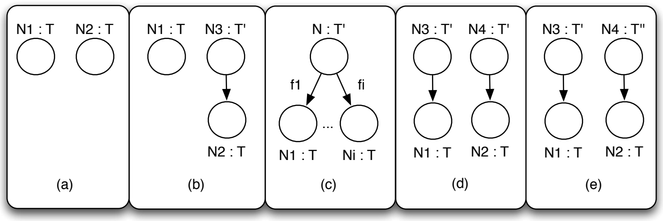


Fig. 11. Comparing nodes using their min-parents.

node of type T to store the minimum element. The conjunction of this axiom and the one generated by procedure *order_root_nodes()* (Fig. 12) forces root nodes to always be smaller than non-root nodes.

Procedure *order_same_min_parent()* (shown in Fig. 14) outputs an axiom that sorts 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 $N_1, \ldots, N_i$.

Procedure *order_same_min_parent_type()* (presented in Fig. 15) creates an axiom that sorts nodes with different min-parents belonging to the same type T'. 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 the axiom sorts N_1 and N_2 following the order between N_3 and N_4.

Finally, the procedure *order_diff_min_parent_types()* shown in Fig. 16 sorts nodes N_1 and N_2 of type T whose min parents have different types. Notice that the axiom orders the nodes following the order in which the classes of the parent nodes were defined in the source Java file.

In order to avoid "holes" in the ordering, procedure *avoid_holes()* (presented in Fig. 17), adds in each signature T a fact stating that whenever a node of type T is reachable in the heap all the smaller ones in the ordering are also reachable.

3.2.1 Symmetry Breaking Predicates: An Example

In order to make the introduction of the symmetry breaking predicates more accessible to the reader, we now present an example. Let us consider the class for red-black trees presented in Fig. 18.

The scopes for analysis will be:

- 1 RBTree atom,
- 5 RBTNode atoms, and
- 5 Integer atoms.

Following procedure *instrument_Alloy()* (Fig. 5), fields `left` and `right` are replaced with fields `fleft` (the forward part of field `left`), `bleft` (the backward part of `left`), `fright` (the forward part of `right`) and `bright` (the backward part of `right`), respectively. Only these two fields are split because these are the only fields that match the definition of recursive field.

The procedure introduces the following axiom to force `fleft+bleft` to be a well–defined total function:

```
for each type $T$ do
  "fact {"
  Let $g_1, \ldots, g_k$ be the root nodes of type $T$
  for $i = 1$ to $k$ do
    for $j = i + 1$ to $k$ do
      "( $g_i \neq$null "
      for $w = i + 1$ to $j - 1$ do
        " and ( $g_w$=null "
        for $v = 0$ to $i$ do
          " or $g_w = g_v$"
        end for
        " ) "
      end for
      " and $g_j \neq$null "
      for $h = 1$ to $i - 1$ do
        " and $g_h \neq g_j$"
        " and $g_h \neq g_i$"
      end for
      ") implies $\langle g_i, g_j \rangle \in nextT$[]"
    end for
  end for
  "}"
end for
```

Fig. 12. The *order_root_nodes()* procedure

```
for each type $T$ do
  "fact {"
  Let $g_1, \ldots, g_k$ be the root nodes of type $T$
  for $i = 1$ to $k$ do
    "( ("
    for $j = 1$ to $i - 1$ do
      "$g_j$=null and "
    end for
    $min_T \leftarrow$ minimum singleton from $T$
    "$g_i \neq$null ) implies $g_i = min_T$ )"
    if $i < k$ then
      " and "
    end if
  end for
  "}"
end for
```

Fig. 13. The *root_is_minimum()* procedure

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

A similar Alloy fact is appended in order to make `fright+bright` a total function.

Our model of Java heaps consists of graphs $\langle N, E, L, R \rangle$. In the present example nodes are the objects from signatures `RBTree`, `RBTNode` and `Integer`, or the value `null`. Labels correspond to field names, and R is the receiver variable `this`, of type `RBTree`.

Algorithm `type_ordering` (see Fig. 6) produces the following order:

```
for each $T, T'$ types do
  Let $f_1, \ldots, f_k$ be the non-recursive and forward
  fields of type $T' \rightarrow T$
  if $k > 1$ then
    "fact {
    all disj o1,o2: $T$ |
      let p1=$minP_T$[o1] |
      let p2=$minP_T$[o2] |
        ( o1+o2 in FReach[] and
        some p1 and some p2 and
        p1=p2 and p1 in $T'$ ) implies ("
    for $i = 1$ to $k - 1$ do
      for $j = i + 1$ to $k$ do
        if $i > 1$ then
          "and"
        end if
        "( ( p1.$f_i$=o1 "
        for $l = i + 1$ to $j - 1$ do
          "and $minP_T$[p1.$f_l$] $\neq$ p1"
        end for
        "and p1.$f_j$=o2 ) implies o2 = o1.nextT[] )"
      end for
    end for
    ")}"
  end if
end for
```

Fig. 14. The *order_same_min_parent()* procedure

```
for each $T, T'$ types do
  if exists a field $f$:$T' \mapsto T$ then
    "fact {
      all disj o1,o2: $T$ |
      let p1=$minP_T$[o1] |
      let p2=$minP_T$[o2] |
      ( o1+o2 in FReach[] and
      some p1 and some p2 and
      p1!=p2 and p1+p2 in $T'$ and p1 in $prevsT'$[p2] )
        implies o1 in $prevsT$[o2]
    }"
  end if
end for
```

Fig. 15. The *order_same_min_parent_type()* procedure

```
for each $T$ type do
  Let $\{T'_i\}$ be the ordered subset of types
  s.t. exist fields $f : T'_j \mapsto T$, $g : T'_k \mapsto T$, $j < k$.
  "fact {
  all disj o1,o2: $T$ |
    let p1=$minP_T$[o1] |
    let p2=$minP_T$[o2] |
    ( o1+o2 in FReach[] and
    some p1 and some p2 and
    p1 in $T'_j$ and p2 in $T'_k$ )
      implies o1 in $prevsT$[o2] )
  }"
end for
```

Fig. 16. The *order_diff_min_parent_types()* procedure

```
for each $T$ type do
  "fact {
  all o: $T$ |
    o in FReach[] implies
      $prevs_T$[o] in FReach[]
  }"
end for
```

Fig. 17. The *avoid_holes()* procedure

```
class RBTNode extends Object {
  RBTNode left;
  RBTNode right;
  Integer value;
  boolean is_black;
}

class RBTree extends Object {
  RBTNode root;
}
```

Fig. 18. A red-black trees class hierarchy

1) RBTree
2) RBTNode
3) Integer

Also, assume that field declarations appear in the following order:

1) root : RBTree $\mapsto$ one (RBTNode+null)
2) fleft : RBTNode $\mapsto$ lone (RBTNode+null)
3) bleft : RBTNode $\mapsto$ lone (RBTNode+null)
4) fright: RBTNode $\mapsto$ lone (RBTNode+null)
5) bright: RBTNode $\mapsto$ lone (RBTNode+null)
6) value : RBTNode $\mapsto$ one (Integer+null)
7) is_black : RBTNode $\mapsto$ one boolean

Executing procedure *local_ordering()* introduces new auxiliary functions. For the example (only for signature `RBTNode`), the procedure outputs:

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

```
fun min_RBTNode [os: set RBTNode] : lone RBTNode {
  os - os.^next_RBTNode[] }
```

```
fun prevs_RBTNode[o : RBTNode] : set RBTNode {
  o.^(~next_RBTNode[]) }
```

Similarly, the procedure outputs function definitions for types `RBTree` and `Integer`.

Procedure *global_ordering()* (Fig. 8) outputs the declaration of function `globalNext`. This function provides an ordering on all objects in the heap. As the reader may notice, each `next_T` is subsumed in `globalNext`.

```
fun globalNext[]: Object -> Object {
  RBTree0->RBTNode0
  + RBTNode0->RBTNode1 + ... + RBTNode3->RBTNode4
  + RBTNode4->Integer0
  + Integer0->Integer1 + ... + Integer3->Integer4
}
```

The following min-parent functions are defined by procedure *define_min_parent()* (Fig. 9). Notice that since there are no fields having objects of type `RBTree` in their range, no `minP_RBTree` function is defined.

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

fun minP_Integer[o: Integer]: Object {
  globalMin[(value).o]
}
```

Procedure *define_freach()* (Fig. 10) yields the definition of a function that characterizes the reachable heap objects:

```
fun FReach[]: set Object {
  this.*(root + value + fleft + fright)
}
```

Notice that field `is_black` is excluded because `boolean` values are not heap objects and the `FReach` function returns a set of heap objects. So far no axioms were introduced other than those constraining the additions of forward and backward fields to be total functions.

Procedure *order_root_nodes()* (Fig. 12) does not output any axioms because there is only one root node, namely, `this`, of type `RBTree`. Procedure *root_is_minimum()* (Fig. 13) outputs:

```
fact { this != null implies this = RBTree_0 }
```

Regarding procedure *order_same_min_parent()* (Fig. 14), since there is only one field from type `RBTree` to type `RBTNode`, there are no two objects with type `RBTNode` with the same min-parent in signature `RBTree`. The same reasoning applies to `RBTNode` and `Integer`. Notice instead that there are two forward fields from type `RBTNode` to type `RBTNode` (namely, `fleft` and `fright`). The axiom produced by *order_same_min_parent()* (described below) orders objects of type `RBTNode` with the same min-parent of type `RBTNode`:

```
fact {
 all disj o1, o2 : RBTNode |
  let p1 = minP_RBTNode[o1] |
  let p2 = minP_RBTNode[o2] |
   ( o1+o2 in FReach[] and
     some p1 and some p2 and
     p1 = p2 and p1 in RBTNode
   ) implies
     ( ( o1 = p1.fleft and o2 = p1.fright) implies
          o2 = o1.next_RBTNode[]
     )
}
```

Procedure *order_same_min_parent_type()* (Fig. 14) yields three axioms. The first one, included below, orders objects of type `RBTNode` with different min-parents of type `RBTNode`. The other two axioms are similar and sort objects of type `Integer` with different `RBTNode` min-parents, and objects of type `RBTNode` with different `RBTree` min-parents. Notice that since $scope(\texttt{RBTree}) = 1$, the last axiom is identically true and can be automatically removed.

```
fact {
 all disj o1, o2 : RBTNode |
  let p1 = minP_RBTNode[o1] |
  let p2 = minP_RBTNode[o2] |
   (o1+o2 in FReach[] and
   some p1 and some p2 and
   p1!=p2 and p1+p2 in RBTNode and
   p1 in prevs_RBTNode[p2])
       implies o1 in prevs_RBTNode[o2]
}
```

Only one type (`RBTNode`) satisfies the conditions required by procedure *order_diff_min_parent_types()* (Fig. 16). In effect, `RBTNode` is the only type for which there are fields pointing to it coming from two different types (for instance, fields `fleft` and `root` have the right typing). The procedure generates the following axiom, which orders objects of type `RBTNode` whose min-parents are one of type `RBTree`, and the other of type `RBTNode`:

```
fact {
  all disj o1, o2 : RBTNode |
    let p1 = minP_RBTNode[o1] |
    let p2 = minP_RBTNode[o2] |
      (o1+o2 in FReach[] and
      some p1 and some p2 and
      p1 in RBTNode and p2 in RBTree)
           implies o1 in prevs_RBTNode[o2]
}
```

Procedure *avoid_holes()* (Fig. 17) outputs the following axiom for signature `RBTNode`:

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

This procedure also generates similar axioms for signatures `RBTree` and `Integer`. Notice that since $scope(\texttt{RBTree}) = 1$, the resulting fact is identically true and is automatically removed.

3.3 A Correctness Proof

Theorem 3.1 below 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. The proof proceeds by renaming nodes following the conditions from Fig. 11, in a way that induces an isomorphism.

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*.

Proof: For each signature T, let $n_{root,T}$ be the number of root objects from T. For each pair of signatures T, T', let $n_{T,T'}$ be the number of objects from T whose min-parent has type T' (notice that although min-parent is not fully defined, we can determine its type due to the linear ordering imposed on signature names). Assign the first $n_{root,T}$ elements from T to root elements. Notice that this satisfies the condition depicted in Fig. 11(a). Use the linear ordering between types and assign, for each signature T', $n_{T,T'}$ objects from T for nodes with min-parent in T'. When doing so, assign smaller objects (w.r.t. the linear ordering `nextT`) to smaller (w.r.t. the linear ordering on signature names) T' signature names. Notice that this satisfies the conditions depicted in Fig. 11(b) and 11(e). It only remains to determine the order between nodes in the same type and whose min-parents have the same type. Follow the directions given in Figs. 11(c) and 11(d). This defines a bijection b between nodes in H and nodes in H'. We still have to label heap arcs. Let n_1, n_2 be nodes in H connected via an edge labeled r. Notice that $b(n_1)$ and $b(n_2)$ have the same type as n_1 and n_2, respectively. Therefore, if r is not recursive, use r as the label for the edge between $b(n_1)$ and $b(n_2)$. If r is recursive, then n_1 and n_2 have the same type or $n_2 = \texttt{null}$, and the same is true for $b(n_1)$ and $b(n_2)$. Thus, since there is a total order on each type, if $b(n_1) < b(n_2)$ or $n_2 = \texttt{null}$ set the label of the edge between $b(n_1)$ and $b(n_2)$ to *fr*. Otherwise, set it to *br*. □

DEFINITION 3.2: Given a heap $H = \langle N, E, L, R\rangle$ for an instrumented model, its *Java-source BFS listing* (JBFS listing for short) is a breadth-first search listing of N that satisfies:

- Root nodes are listed first, following the order they were declared in the Java source file.
- Given an already listed node n, its children are listed according to the order in which the fields pointing to them were declared in the Java source file.

Notice that Def. 3.2 defines the BFS listing uniquely, and therefore, two heaps H, H' are different iff their JBFS listings are different.

DEFINITION 3.3: Given a heap $H = \langle N, E, L, R\rangle$ for an instrumented model, its *min-parent sub-heap* (denoted by SH) is the restriction of H obtained by only keeping those arcs satisfying $n_1 \to n_2$ iff $n_1 = minP[n_2]$.

LEMMA 3.4: Let H be a heap for an instrumented model. Then, SH is a forest.

Proof: Notice that all nodes have at most one incoming arc. Only the root nodes do not have incoming arcs. Let us show that SH is a forest (set of trees) by showing that there are no cycles. First, homogeneous cycles (those whose nodes all have the same type) cannot exist because forward recursive fields relate nodes with greater nodes. Second, heterogeneous cycles (those involving nodes from at least two different types) cannot exist because (see the paragraph after the definition of function *globalMin*) heterogeneous cycles in H must have an input arc $a \to b$ in which a's type is smaller than b's type. Therefore, $minP[b] = a$, which is outside the cycle. □

LEMMA 3.5: Let H and H' be isomorphic heaps for an instrumented model. Then, SH and SH' are also isomorphic.

Proof: Let $i : H \to H'$ be an isomorphism. Let us prove that i is an isomorphism from SH to SH'. Let $n_1, \ldots, n_i, \ldots$ and $m_1, \ldots, m_i, \ldots$ be the JBFS listings of SH and SH', respectively. If SH and SH' are not isomorphic, there must exist a minimum index j such that $i(n_j) \neq m_j$. But, by definition of JBFS listing, one of the following conflictive situations must occur:

1) n_j is a root node, (and then $i(n_j) = m_j$), or
2) since by Lemma 3.4 SH and SH' are forests, there is a unique node n_k (with $k < j$) and a field f such that $n_k.f = n_j$. But then, by j's minimality, it must be $i(n_k) = m_k$. Since i is an isomorphism, it must be $i(n_j) = i(n_k.f) = m_k.f = m_j$.

□

LEMMA 3.6: Let H and H' be isomorphic heaps for an instrumented model. Let $n_1, \ldots, n_i, \ldots, n_j, \ldots$ and $m_1, \ldots, m_i, \ldots, m_j, \ldots$ be the JBFS listings of SH and SH', respectively. Then,

$$n_i < n_j \quad \text{iff} \quad m_i < m_j \ .$$

Proof: Let us assume the property is false and let us arrive at a contradiction. Since the property is false, there must exist a minimum i_0 such that $n_{i_0} < n_j$ and $m_{i_0} > m_j$. Similarly, let j_0 be such that it is minimum among the values of j. Thus, $n_{i_0} < n_{j_0}$ and $m_{i_0} > m_{j_0}$. We will consider the following cases:

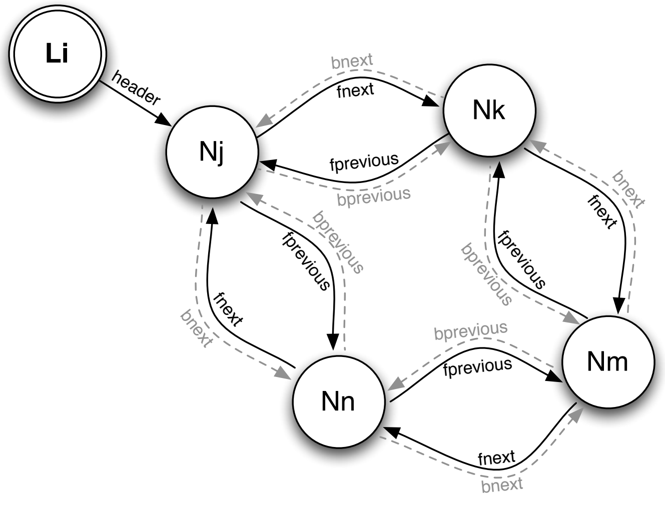
1) n_{i_0} and n_{j_0} are root nodes: The contradiction is immediate because root nodes are explicitly ordered by axiom order_root_nodes (Fig. 12).
2) n_{i_0} is a root node and n_{j_0} is not: Due to the isomorphism between SH and SH' (Lemma 3.5), m_{i_0} must be a root node and m_{j_0} must not be a root node. Axiom root_is_minimum (Fig. 13) explicitly establishes that $m_{i_0} < m_{j_0}$.
3) n_{i_0} and n_{j_0} have the same min-parent n_k, and the field pointing to n_{i_0} is declared before the field pointing to n_{j_0}: Due to the isomorphism between SH and SH' (Lemma 3.5), it must be $i(n_{i_0}) = m_{i_0}$ and $i(n_{j_0}) = m_{j_0}$. Similarly, it must be $i(n_k) = m_k$. Axiom order_same_min_parent (Fig. 14) explicitly establishes that $m_{i_0} < m_{j_0}$.
4) n_{i_0} and n_{j_0} have min-parents of different types, and $minP[n_{i_0}]$'s type is less than $minP[n_{j_0}]$'s type: Due to the isomorphism between SH and SH' (Lemma 3.5), it must be $minP[m_{i_0}]$'s type less than $minP[m_{j_0}]$'s type. Thus, by axiom order_diff_min_parent_types (Fig. 16), $m_{i_0} < m_{j_0}$.
5) n_{i_0} and n_{j_0} have different min-parents of the same type: By axiom order_same_min_parent_type (see Fig. 15), it must be $minP[n_{i_0}] < minP[n_{j_0}]$ and $minP[m_{i_0}] > minP[m_{j_0}]$. Since the index of $minP[n_{i_0}]$ is less than i_0, the minimality of i_0 is violated.

□

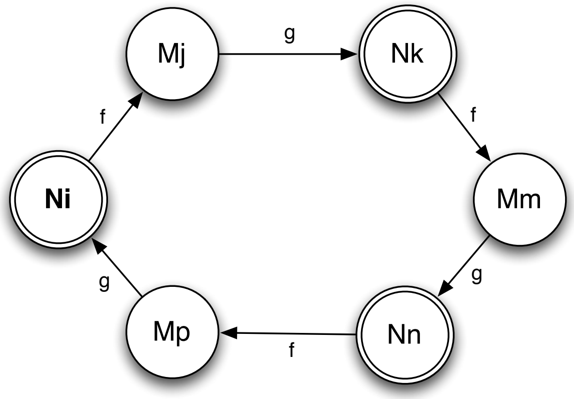
Theorem 3.7 below shows that the instrumentation indeed yields a canonicalization of the heap. The intuition behind the proof is that the heap is characterized by its min-parent sub-heap. Therefore, canonicity follows from proving that isomorphic heaps have the same JBFS listings of their min-parent sub-heaps.

THEOREM 3.7: Let H, H' be heaps for an instrumented model. If H is isomorphic to H', then $H = H'$.

Proof: Since H and H' are isomorphic it suffices to show that the JBFS listings of SH and SH' are the same. If this is not the case, there must be a minimum position



a) A circular doubly-linked list with the TACO instrumentation.



b) An heterogenous cycle.

Fig. 19. Breaking symmetries in two cyclic heaps.

i_0 where the listings differ. Let $n_1, n_2, \ldots, n_{i_0}, \ldots, n_i, \ldots$ be the listing of SH, and $m_1, m_2, \ldots, m_{i_0}, \ldots, m_i, \ldots$ be the listing of SH'. Since i_0 is minimal, it must be $n_1 = m_1$, $n_2 = m_2$, …, $n_{i_0-1} = m_{i_0-1}$. Moreover, let us assume without loss of generality that $n_{i_0} > m_{i_0}$. Let $j > i_0$ such that

$$n_{i_0} > n_j \text{ and } m_{i_0} < m_j \ .$$

Such j exists because by axiom avoid_holes (Fig. 17) there are no holes in the listings. This contradicts Lemma 3.6. □

Notice that class fields may induce cycles in the heap, and even induce indirect cycles (think for instance of fields $f_1 : T_1 \to T_2$, $f_2 : T_2 \to T_1$). In Fig. 19 we present two cyclic heaps. In the following paragraphs we will explain how symmetries are broken in these examples.

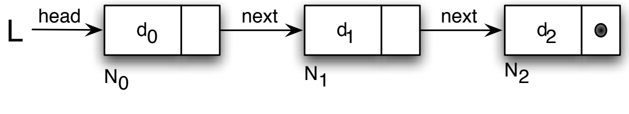
EXAMPLE 3.8: Let us analyze first the heap depicted in Fig. 19.a). This heap configuration corresponds to a circular doubly-linked list from the Apache package `commons.collections` (one of the benchmark classes we will use in Sec. 5). Node L_i is the receiver object *this*, whose type is `AbstractLinkedList`. Nodes N_j, N_k, N_m and N_n have type `LinkedNode`. Algorithm *type-ordering* sets `CyclicList` < `LinkedNode`. Since there is exactly one node with type `CyclicList`, by axiom *avoid_holes* it has to be node L_0. Notice that for all nodes of type `LinkedNode` but N_j, their `min_parent` has type `LinkedNode`. Since $N_j's$ `min_parent` has type `CyclicList`, by axiom *order_diff_min_parent_types* must be $j < k$, $j < m$ and $j < n$. Then, $\texttt{minP}[N_k] = \texttt{minP}[N_n] = N_j$. Assuming that field *next* was declared before field *previous*, by axiom *order_same_min_parent* must be $k < n$. Let us compare now indices n and m. $\texttt{minP}[N_n] = N_j$ and (since $k < n$) $\texttt{minP}[N_m] = N_k$. Since $j < k$, by axiom *order_same_min_parent_type*, must be $n < m$. We then have $j < k < n < m$. By axiom *avoid_holes* must be $j = 0$, $k = 1$, $n = 2$ and $m = 3$.

EXAMPLE 3.9: Let us analyze the heap depicted in Fig. 19.b). Node N_i is the receiver object *this*. N_i, N_k and N_n have type T_1. Nodes M_j, M_m and M_p have type T_2. Algorithm *type-ordering* sets $T_1 < T_2$. Since *this* is always a root, by axiom *root_is_minimum* is $i < k$ and $i < n$. Since $\texttt{minP}[M_j] = N_i$, $\texttt{minP}[M_m] = N_k$ and $\texttt{minP}[M_p] = N_n$, by axiom *order_same_min_parent_type* is $j < m$ and $j < p$. Let us compare now nodes N_k and N_n. $\texttt{minP}[N_k] = M_j$ and $\texttt{minP}[N_n] = M_m$. Since $j < m$, by axiom *order_same_min_parent_type*, must be $k < n$. Thus, $i < k < n$, and by axiom *avoid_holes* is $i = 0$, $k = 1$ and $n = 2$. A similar reasoning allows us to conclude that $j = 0$, $m = 1$ and $p = 2$.

4 COMPUTING TIGHT BOUNDS

A distinguishing feature of Alloy's backend, KodKod, is that it enables the prescription of partial instances in models. Indeed, each Alloy 4 field `f` is translated to a matrix of propositional variables as described in Fig. 3, together with two *bounds* (relation instances) $L_{\texttt{f}}$ (the lower bound) and $U_{\texttt{f}}$ (the upper bound). As we will see, these bounds provide useful information. Consider for instance relation `next` from the singly-linked list model presented in Sec. 2. If a tuple $\langle N_i, N_j\rangle \notin U_{\texttt{next}}$, then no instance of field `next` can contain $\langle N_i, N_j\rangle$, allowing us to replace p_{N_i,N_j} in $M_{\texttt{next}}$ (the matrix of propositional variables associated with relation `next`) by the truth value *false*. Similarly, if $\langle N_i, N_j\rangle \in L_{\texttt{next}}$, 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} with 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.

Assume that the class invariant for representing a singly linked list requires lists to be acyclic. Assume also that nodes have identifiers $N_0, N_1, N_2, \ldots$. Thus, a list instance will have the shape



Notice that since lists are assumed to be acyclic, it is easy to see that some tuples are deemed to never be contained in any `next` relation instance. Since no node may refer to itself, there is no instance such that any of tuples $\langle N_0, N_0\rangle$, $\langle N_1, N_1\rangle$ and $\langle N_2, N_2\rangle$ are contained in relation `next`. If we could determine this before translating to a propositional formula, then these tuples could be safely removed from the U_{next} upper bound. By doing so, propositional variables representing membership of these tuples (namely, p_{N_0,N_0}, p_{N_1,N_1} and p_{N_2,N_2}) could be replaced with value *false*, leading to a formula with fewer 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 Sec. 5) 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 $U_{\texttt{next}}$ 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. In this section we concentrate on how to determine if a given pair can be removed from an upper bound relation, therefore improving the analysis performance.

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 this section we made reference to the lower and upper bounds ($L_{\texttt{f}}$ and $U_{\texttt{f}}$) attached to an Alloy field `f` during its translation to a KodKod model. For the rest of this section, we use the term *bound* to refer to the upper bound $U_{\texttt{f}}$.

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 the class invariant for the red-black tree structure we introduced in Sec. 3 states 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 always 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).

In particular, these assumptions may be fulfilled by using the symmetry breaking predicates introduced in Sec. 3. Following the breadth-first order heap canonization, given a tree composed of nodes $N_0, N_1, \ldots, N_k$, node N_0 is the tree root, $N_0.\mathit{left} = N_1$, $N_0.\mathit{right} = N_2$, etc. Observe that the breadth-first ordering allows us to impose more constraints on the structure. For instance, it is no longer possible that $N_0.\mathit{left} = N_2$. 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 beginning of this section, 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 Sec. 5, for a scope of 10 red-black tree nodes, this analysis allows us to reduce the number of propositional variables from 650 to 200.

The usefulness of the previous reasonings strongly depends on the following two requirements:

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 $U_{\texttt{f}}$.

To cope with requirement 1 we will rely on the symmetry breaking predicates we introduced in Sec. 3. With respect to requirement 2, in Sec. 4.1 we will present a fully automatic and effective technique for checking feasibility.

4.1 Symmetry Breaking and Tight Bounds

In the previous section we discussed the representation of red-black trees. 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 and to the class invariant 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 $U_{\texttt{F}}$, TACO proceeds as follows:

1) Synthesizes the instrumented model following the procedure shown in Sec. 3.

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 analysis:

```
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`->`N_j `in F` satisfying the class invariant. Therefore, the edge is infeasible within the provided scope. It is then removed from $U_{\texttt{F}}$, the upper bound relation associated with 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.

All these analyses are independent. A naive algorithm to determine feasibility consists of performing all the checks in parallel. Unfortunately, the time required for each one of these analyses is highly irregular. Some of the checks take milliseconds, and others may exhaust available resources while searching for the complex instances that have to be produced.

The algorithm for bound refinement we used in [26] (whose pseudocode is given in Fig. 20), is an iterative procedure that 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 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 into three sets: unsatisfiable checks, satisfiable 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, giving rise to the next iteration. This process, 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 faults.

THEOREM 4.1: Let H be a memory heap exposing a fault. 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 $U_{\texttt{g}}$.

Proof sketch: Let H' be the heap from Theorem 3.1. It satisfies the instrumentation and, since H' is isomorphic to H, it also exposes the fault. Assume there is in H' an edge $N_i \to N_j$ labeled `g`, such that $N_i \to N_j \notin U_{\texttt{g}}$. Since during code analysis TACO includes the class invariant as part of the precondition, heap H' must satisfy the invariant. But since $N_i \to N_j \notin U_{\texttt{g}}$, the Alloy analysis

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

must have returned UNSAT. Then, there is no memory heap that satisfies the invariant and contains the edge $N_i \to N_j$, leading to a contradiction.

```
global $TIMEOUT$

function $fill\_queue(upper\_bounds, spec)$ : int
int $task\_count = 0$
for each edge $A \xrightarrow{f} B$ in $upper\_bounds$ do
    $M := create\_Alloy\_model(A \xrightarrow{f} B, upper\_bounds, spec)$
    $task\_count$++
    $ENQUEUE(< A \xrightarrow{f} B, M >, workQ)$
end for
return $task\_count$

function $ITERATIVE\_MASTER(scope, spec) : upper\_bounds$
$workQ := CREATE\_QUEUE()$
$upper\_bounds := initial\_upper\_bounds(spec, scope)$
repeat
    $task\_count := fill\_queue(upper\_bounds, spec)$
    $result\_count := 0$
    $timeout\_count := 0$
    $unsat\_count := 0$
    while $result\_count$ != $tasks\_count$ do
        $< A \xrightarrow{f} B, analysis\_result > := RECV()$
        $result\_count$++
        if $analysis\_result == UNSAT$ then
            $upper\_bounds := upper\_bounds - A \xrightarrow{f} B$
        else if $analysis\_result == TIMEOUT$ then
            $timeout\_count$++
        end if
    end while
    if $unsat\_count == 0$ then
        return $upper\_bounds$
    end if
until $timeout\_count == 0$
return $upper\_bounds$

function $ITERATIVE\_SLAVE()$
while size$(workQ) > 0$ do
    $< A \xrightarrow{f} B, M >$ := DEQUEUE$(workQ)$
    $analysis\_result$ := run_stoppable_Alloy$(M, TIMEOUT)$
    SEND$(master, < A \xrightarrow{f} B, analysis\_result >)$
end while
```

Fig. 20. TACO's algorithm for generational bound refinement.

For most of the case studies we report in Sec. 5 it was possible to check all edges using this algorithm. Since bounds only depend on the class invariant, the signatures scopes and the typing of the method under analysis, the same bound is used (as will be seen in Sec. 5) to improve the analysis of different methods. By extending TACO's architecture, a bound, once computed, is stored in a bounds repository as shown in Fig. 21.

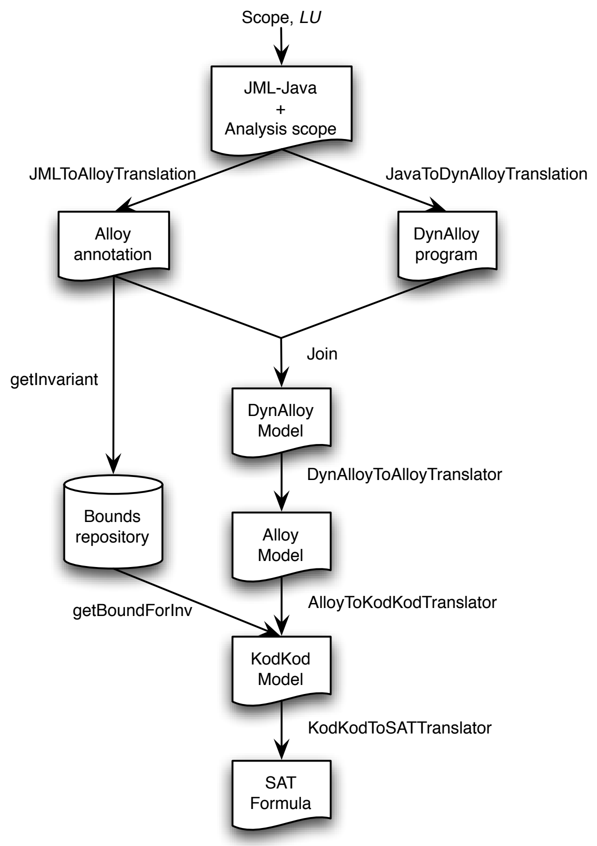


Fig. 21. TACO architecture extended with a bounds repository.

It is generally the case that the number of processors is significantly smaller than the number of analyses that can be run in parallel. As we have already mentioned, analysis time for feasibility checks is highly irregular. Thus, by the time an analysis is allocated to a given processor, verdicts from previous edges may have been already reported. In the TACO algorithm presented in Fig. 20 a *generational* approach is taken. This means that although an UNSAT verdict is known for a given edge, this information has no effect before the current iteration is finished.

An alternative approach for computing bounds is to make use of UNSAT information as soon as it is available. This leads to a third algorithm, shown in Fig. 22. For the rest of this article, we will refer to this alternative algorithm as the *eager* algorithm. The main characteristic of this algorithm is that upper bounds are updated as soon as an UNSAT certificate is obtained. Therefore, Alloy models being allocated for analysis make use of the most recent upper bound information. Also, since the Alloy Analyzer outputs a model whenever a feasibility check returns SAT, the algorithm marks as satisfiable all variables corresponding to edges that are reachable in that model from the root nodes. This improves the efficiency of the tool by avoiding the analysis of those edges.

```
global $TIMEOUT, upper\_bounds, edgeQ$

function $DYNAMIC\_MASTER()$ :
$edgeQ := CREATE\_QUEUE()$
$upper\_bounds :=$ initial_upper_bounds($spec, scope$)
for $A \xrightarrow{f} B$ in $upper\_bounds$ do
    ENQUEUE($edgeQ, A \xrightarrow{f} B$)
end for

function $DYNAMIC\_SLAVE()$ :
while size($edgeQ$) $> 0$ do
    $< A \xrightarrow{f} B, M > :=$ DEQUEUE($edgeQ$)
    $M :=$ create_Alloy_Model($A \xrightarrow{f} B, upper\_bounds, spec$)
    $< analysis\_result, I > :=$ run_stoppable_Alloy($M, TIMEOUT$)
    if $analysis\_result == UNSAT$ then
        $upper\_bounds := upper\_bounds - A \xrightarrow{f} B$
    else if $analysis\_result == SAT$ then
        for $A' \xrightarrow{f'} B'$ in $I$ do
            remove($edgeQ, A' \xrightarrow{f'} B'$)
        end for
    else
        ENQUEUE($A \xrightarrow{f} B, edgeQ$)
    end if
end while
```

Fig. 22. TACO's algorithm for dynamic bound refinement.

5 Experimental Results

In this section we report the results obtained from conducting several experiments. We analyze seven collection classes with increasingly complex class invariants. Using these classes we will study the performance of TACO in several ways. We will denote by TACO$^-$ the translation implemented in TACO, but without the symmetry reduction axioms or the tight bounds. In Sec. 5.1 we study the effect that the inclusion of the symmetry breaking predicates has on the analysis time. This is achieved by comparing TACO$^-$ with TACO. In Sec. 5.2, we compare the parallel algorithms for computing bounds presented in Figs. 20 and 22. Section 5.3 reports on the impact of using tighter bounds. Finally, in Secs. 5.4 and 5.5, we compare TACO with several tools in two settings. The first one is a comparison with JForge [19] (a state-of-the-art SAT-based analysis tool developed at MIT). Since the classes we analyze are correct[2], this allows us to compare the tools in a situation where the state space must be exhausted. The second one is when we study the error-finding capabilities of TACO against several state-of-the-art tools based on SAT-solving, model checking and SMT-solving.

2. Actually, as we will show in Sec. 5.5.3, there is a fault in one implementation that has not been reported before.

Experimental Setup

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:** A caching circular double linked list implementation of interface `List` from the Apache package `commons.collections`.
- **BSTree:** A binary search tree implementation from [45]
- **TreeSet:** The implementation of class `TreeSet` from package `java.util`, based on red-black trees.
- **AVL:** An implementation of AVL trees obtained from the case study used in [4].
- **BHeap:** An implementation of binomial heaps used as part of a benchmark in [45].

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 classes `TreeSet`, `AVL` and `BSTree` 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. We also analyze a method for extracting the minimum element from a binomial heap, that contains a previously unknown fault (we discus it extensively in Sec. 5.5).

Loops are unrolled up to 10 times, and no contracts for called methods are used (we inline their code). 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".

The parallel algorithms for computing bounds were 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. Those experiments for which there exists a non-deterministic component in the behavior of the algorithm were run ten times and the value reported corresponds to the average of all execution times.

5.1 Analysis Using Symmetry Breaking Predicates

As mentioned before, none of the main contributions of this article were implemented in TACO$^-$. In this sense, the analysis time of TACO$^-$ can be used as a reference value for measuring the improvement produced by the inclusion of symmetry breaking predicates as well as by the use of tight bounds.

In Table 1 we compare the analysis time of TACO$^-$ against a version of TACO that only adds the symmetry breaking predicates (we will call this intermediate version TACO^{sym}). In other words, bounds are neither computed nor used by TACO^{sym}. The cell highlighting denotes which tool needed the smaller amount of computing time. If both tools required the same amount of computing time or both tools reached the time limit, no cell was highlighted.

#Node			5	7	10	12	15	17	20
LList	contains	T^-	00:03	00:05	00:08	00:11	00:13	00:22	00:34
		T^s	00:00	00:01	00:05	00:09	00:25	00:46	00:50
	insert	T^-	00:05	00:27	TO	TO	TO	TO	TO
		T^s	00:01	00:01	00:06	00:10	00:29	00:39	01:46
	remove	T^-	00:04	00:09	01:14	00:33	04:26	01:25	02:57
		T^s	00:01	00:01	00:02	00:03	00:07	00:22	00:38
AList	contains	T^-	00:05	00:11	00:29	00:38	00:42	01:37	01:21
		T^s	00:01	00:04	00:32	00:45	02:22	07:46	243:54
	insert	T^-	00:06	00:14	11:25	347:39	TO	TO	TO
		T^s	00:01	00:03	00:16	00:38	03:21	15:08	TO
	remove	T^-	00:04	00:05	01:02	26:22	TO	TO	TO
		T^s	00:01	00:01	00:02	00:05	01:00	04:49	258:21
CList	contains	T^-	00:46	03:51	00:22	01:01	01:30	06:39	01:09
		T^s	00:01	00:06	00:25	01:48	04:50	18:18	TO
	insert	T^-	02:43	TO	TO	TO	TO	TO	TO
		T^s	00:06	00:29	02:29	06:52	31:48	112:25	TO
	remove	T^-	00:11	22:22	TO	TO	TO	TO	TO
		T^s	00:01	00:01	00:04	00:18	03:06	12:17	TO
BSTree	contains	T^-	16:30	320:39	TO	TO	TO	TO	TO
		T^s	00:03	00:50	136:15	TO	TO	TO	TO
	insert	T^-	TO	TO	TO	TO	TO	TO	TO
		T^s	09:26	128:52	TO	TO	TO	TO	TO
	remove	T^-	02:07	TO	TO	TO	TO	TO	TO
		T^s	00:02	00:35	54:42	TO	TO	TO	TO
TreeSet	contains	T^-	02:13	276:49	TO	TO	TO	TO	TO
		T^s	00:02	00:16	05:35	22:00	186:17	TO	TO
	insert	T^-	21:38	TO	TO	TO	TO	TO	TO
		T^s	01:05	10:48	TO	TO	TO	TO	TO
	remove	T^-	33:18	TO	TO	TO	TO	TO	TO
		T^s	00:56	07:26	216:56	TO	TO	TO	TO
AVL	find	T^-	00:14	27:06	TO	TO	TO	TO	TO
		T^s	00:01	00:01	00:14	01:27	06:24	74:58	TO
	findMax	T^-	00:02	00:04	46:12	TO	TO	TO	TO
		T^s	00:02	00:17	03:30	11:25	177:17	TO	TO
	insert	T^-	01:20	335:51	TO	TO	TO	TO	TO
		T^s	00:11	01:05	16:51	64:01	TO	TO	TO
	remove	T^-	60:53	TO	TO	TO	TO	TO	TO
		T^s	01:09	05:07	80:04	225:50	TO	TO	TO
BHeap	findMin	T^-	00:12	11:41	TO	TO	TO	TO	TO
		T^s	00:01	00:04	00:20	00:52	05:15	14:32	45:47
	decKey	T^-	05:36	TO	TO	TO	TO	TO	TO
		T^s	00:11	01:30	34:36	120:39	TO	TO	TO
	insert	T^-	22:46	391:10	TO	TO	TO	TO	TO
		T^s	00:40	02:23	31:15	91:07	250:09	TO	TO

TABLE 1
Comparison of code analysis times for 10 loop unrolls using TACO$^-$(T$^-$) and TACO^{sym} (T^s).

Table 2 shows the improvement of using the symmetry breaking predicates discussed in Section 3. All methods under analysis are correct with respect to their specification. The first column shows the maximum scope for which TACO$^-$achieves the analysis within the time threshold of 10 hours. Similarly, the second column shows the same information for TACO^{sym}.

Let s be the maximum scope for which both TACO^{-}and TACO^{sym} completed the analysis within the time limit. The third and fourth columns show the analysis times for both tools in that particular scope. Finally, the last column shows the ratio between the time required by TACO^{sym} and TACO^{-}at scope s. As in Table 1, we distinguish the tool that reached the larger scope of analysis as well as the one that required less analysis time by highlighting the corresponding cells.

		max T^{-} scope	max T^{s} scope	T^{-} time	T^{s} time	T^{s} time / T^{-} time
LList	contains	20	20	00:34	00:51	150%
	insert	8	20	07:24	07:34	102.25%
	remove	20	20	02:17	00:38	21.47%
AList	contains	20	20	01:20	243:54	18292%
	insert	13	19	283:08	00:38	0.22%
	remove	13	20	150:57	00:08	0.09%
CList	contains	20	19	00:39	116:40	17948%
	insert	7	18	148:20	00:26	0.29%
	remove	9	19	123:07	00:04	0.05%
BSTree	contains	7	11	320:39	00:49	0.25%
	insert	3	7	02:38	00:08	5.06%
	remove	6	11	325:12	00:06	0.03%
TreeSet	contains	7	16	127:09	00:16	0.21%
	insert	6	7	412:11	04:51	1.19%
	remove	6	10	236:32	02:32	1.07%
AVL	find	11	18	472:33	00:33	0.12%
	findMax	7	15	21:00	00:16	0.127%
	insert	7	14	245:35	01:05	0.44%
	remove	6	12	107:23	02:13	2.07%
BHeap	findMin	8	20	202:54	00:05	0.04%
	decKey	7	12	560:34	01:29	0.26%
	insert	7	15	70:53	02:23	0.36%

TABLE 2
Improvement produced by using the symmetry breaking predicates.

Observe that in most cases TACO^{sym} outperforms TACO^{-} both in maximum scope for which the analysis ends within the time limit and in the amount of time spent in analysis for the maximum scope for which both tools finish. This can be seen in the fifth column corresponding to the analysis times ratio. To summarize the information of the table, 96% of cases show an increase of the maximum scope of analysis, while only for 1 case (4%) this value decreases. This was calculated on the basis of those cases where at least one of the tools reached the timeout limit. Considering all the experiments in the benchmark, TACO^{sym} increases the scope of analysis in 6.57 nodes on average. When comparing the largest common scope for which both tools finish the analysis within the time limit, over 80% of the experiments show a dramatic decrease in the analysis time. When calculating over these cases, the time required by TACO^{sym} to accomplish the analysis is, on average, only 1.85% of the time consumed by TACO^{-}.

5.2 Computing Tight Bounds

In Sec. 4 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 $U_{\texttt{f}}$ is readily built. We will call the union of the upper bounds over all fields, the *upper bound*. In Table 3 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 4.1 starting from the initial upper bound. Given a field f, the instance of U_f that contains all tuples is called *initial upper bound*. The time required to build the initial upper bound is negligible.
3) The time required by the iterative algorithm in Fig. 20 to build the tight upper bound.
4) The time required by the eager algorithm in Fig. 22 to build the same tight upper bound.

Again, we distinguish the algorithm that consumed the smaller amount of time by highlighting the corresponding cell. For both algorithms, the initial timeout used during bound refinement for the individual analyses was set to 2′.

#Node		5	7	10	12	15	17	20
LList	#UB	30	56	110	156	240	306	420
	#TUB	9	13	19	23	29	33	39
	Time I	00:11	00:14	00:23	00:36	01:01	01:23	02:25
	Time E	00:11	00:11	00:15	00:24	00:47	01:04	01:37
AList	#UB	76	128	252	344	512	676	904
	#TUB	33	47	68	82	103	117	138
	Time I	00:16	00:25	00:51	01:26	02:47	09:28	TO
	Time E	00:11	00:14	00:51	00:33	00:55	03:15	300:20
CList	#UB	328	384	498	594	768	904	1138
	#TUB	97	127	172	210	240	277	322
	Time I	00:57	01:13	01:45	02:25	05:27	21:31	575:00
	Time E	00:35	00:46	01:11	01:38	01:46	05:16	TO
BSTree	#UB	90	168	330	468	720	918	1260
	#TUB	54	97	184	257	389	492	669
	Time I	00:22	00:34	01:04	01:46	03:19	05:32	25:10
	Time E	00:11	00:11	00:16	00:38	01:56	04:05	21:19
TreeSet	#UB	170	280	650	852	1200	2006	2540
	#TUB	59	107	200	279	424	533	720
	Time I	00:49	01:13	03:03	05:11	11:30	44:23	97:04
	Time E	00:16	00:30	01:44	02:51	05:19	16:42	40:37
AVL	#UB	150	280	650	852	1200	2006	2540
	#TUB	55	98	177	251	389	491	669
	Time I	00:33	00:57	03:26	09:53	22:03	101:31	579:40
	Time E	00:17	00:32	01:55	03:46	10:36	47:25	168:23
BHeap	#UB	222	360	803	1053	1488	2394	2540
	#TUB	75	123	218	293	423	481	669
	Time I	00:44	01:12	04:00	06:48	20:13	62:50	211:20
	Time E	00:22	01:12	02:46	04:41	10:32	34:50	117:20

TABLE 3
Analysis time in `mmm:ss` for discovering tighter upper bounds using each algorithm.

Table 3 shows that, on average, over 70% of the variables in the bounds can be removed. Let us now compare the performance of computing a tight bound by using the iterative algorithm (Fig. 20) and the eager algorithm (Fig. 22). Observe that, on average, a speed-up of approximately 1.95x is achieved by using the eager algorithm instead of the iterative algorithm for computing bounds. Both iterative and eager algorithms exceeded the 10 hour barrier for only one experiment (cyclic linked list and cache linked list respectively, both for a scope of 20).

Although the aforementioned savings are indeed significant, it is worth mentioning that they fail to achieve a major improvement in asymptotic terms. Figs. 23 and 24 are introduced as two representative cases of the comparison of both algorithms. As these figures illustrate, projections of the same data on a logarithmic scale on the y-axis reveal some interesting offset shifts, yet there is hardly any impact on the slopes.

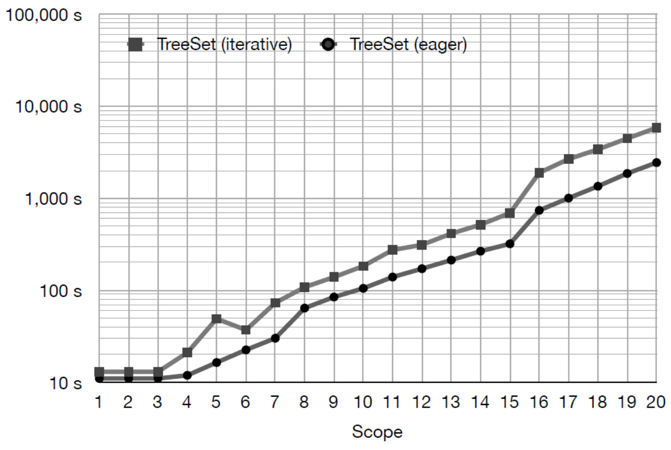


Fig. 23. Analysis time (in logarithmic scale) for computing bounds of TreeSet using the iterative and the eager algorithms

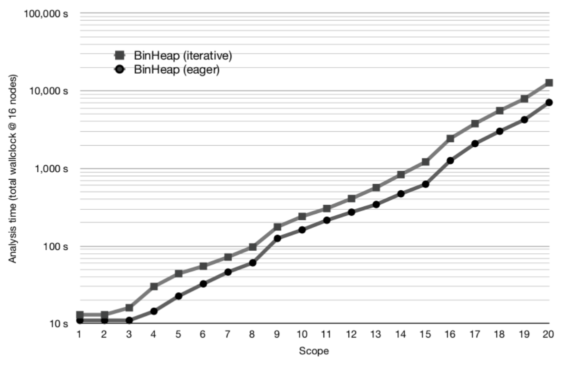


Fig. 24. Analysis time (in logarithmic scale) for computing bounds of Binomial heap using the iterative and the eager algorithms

Both techniques suffer from a high number of aborted partial analysis. We are currently developing strategies to mitigate this problem. We hope that this will help us in devising a more scalable algorithm for computing tight bounds.

5.3 Analyzing the Impact of Using Bounds

In this section we will show the results of systematically tightening the bounds to determine the effects of such change in the SAT-solver behavior. Our hypothesis is that most times a tighter bound leads to a smaller analysis time.

In order to study the effect of tightening the bound, we ran the same analyses varying only this parameter. Up to this point, we have referred to two kinds of bounds: the initial bound (all tuples) and the tightest bound (computed by the distributed algorithms). To evaluate the impact of using bounds we built several approximations ranging from the initial bound to the tightest bound. We produce a bound $B_{n\%}$ by keeping those edges whose feasibility check was reported as UNSAT, and fall within the $n\%$ of the less expensive checks in terms of analysis time. Given two edges e_1 and e_2, we say that e_1 is less expensive than e_2 if the time needed for obtaining a verdict for the feasibility of e_1 is less than that of e_2. Notice that, using this definition, the $B_{100\%}$ bound corresponds to the tightest bound, while the $B_{0\%}$ bound corresponds to the initial bound.

By using the stored logging information from running the distributed algorithm we built the following bounds: $B_{10\%}$, $B_{20\%}$, $B_{30\%}$, $B_{40\%}$, $B_{50\%}$, $B_{60\%}$, $B_{70\%}$, $B_{80\%}$, and $B_{90\%}$.

The reader may notice that computing bounds of different precisions only makes sense when the iterative algorithm for computing bounds (Fig. 20) is used. This is because in the dynamic algorithm of Fig. 22 the analysis time for a given check is strongly influenced by the initial scheduling.

Once the bounds were defined for each collection class, we re-ran each experiment varying the bound. The timeout was set again to 10 hours. We fixed the scope of each method under analysis to be the maximum value such that TACO (using any incremental bound) successfully completed the analysis within the time limit. The rationale behind this decision is to examine the effect on the hardest problems.

Due to the small analysis times, the case studies corresponding to class `LList` were explicitly excluded from this assessment. For the remaining 19 methods under analysis, 8 exhibited an almost strictly monotonic decrease in the analysis time required as the bound got tighter. The improvement is shown in logarithmic scale in Fig. 25.

For the 8 methods under analysis shown in Fig. 26, a dramatic decrease in analysis time is also exhibited. Although some oscillations do occur for a couple of cases, the gain obtained from tightening the bound is clear.

Finally, for the 3 methods shown in Fig. 27 no improvement appears to be obtained by increasing the bound precision. These cases represent the 13% of all methods under analysis. On the contrary, the remaining 87% do exhibit an exponential improvement. Therefore, we conclude that the analysis of the selected benchmark is sensitive to tightening the bounds.

It is worth mentioning that, for those methods that do exhibit an improvement in the analysis as the bound

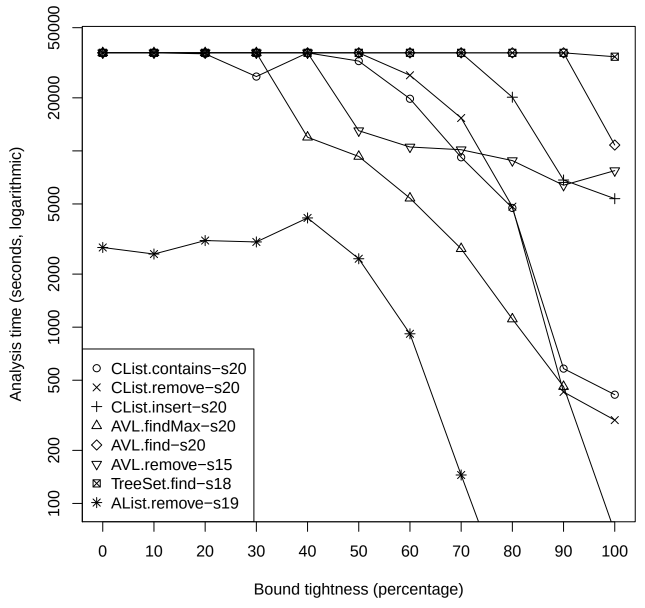


Fig. 25. Analysis time as bound precision is increased.

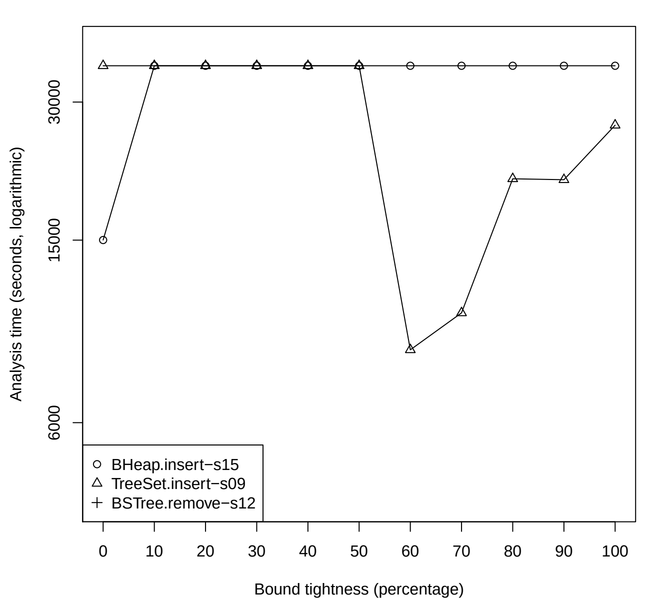


Fig. 27. Analysis time as bound precision is increased.

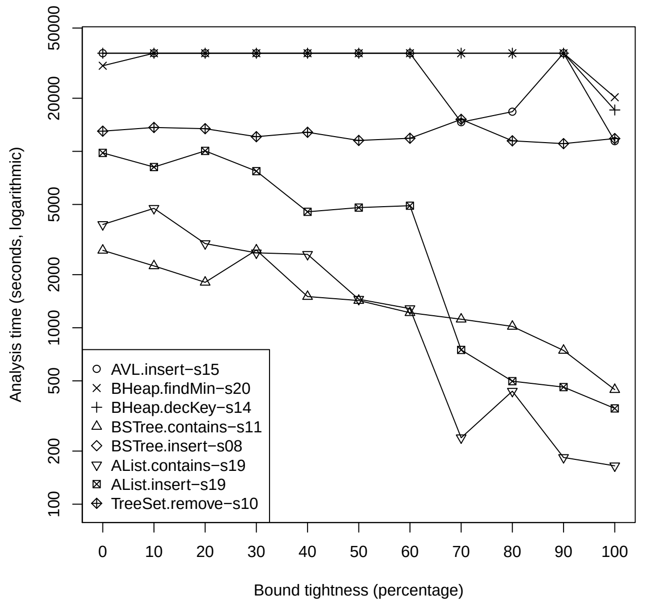


Fig. 26. Analysis time as bound precision is increased.

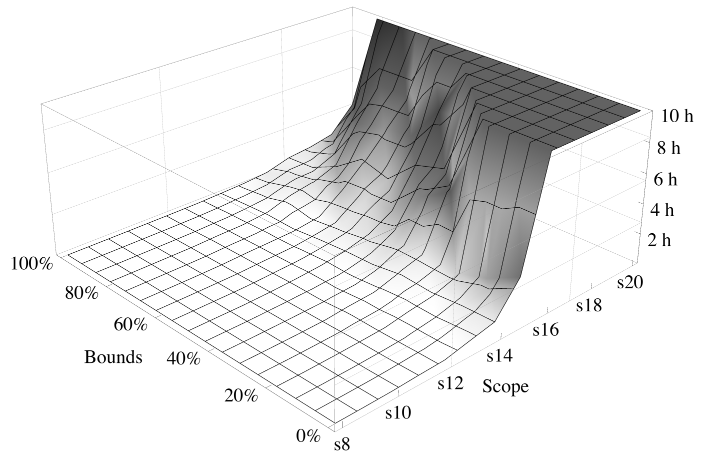


Fig. 28. Analysis time for method *insert* of AVL as scope and bound tightness grows.

precision grows, this improvement also occurs in smaller scopes. To illustrate these improvements to the reader, we also report the results of the analysis times for the method *insert* for the AVL tree and for the cached cyclic linked list. Figs. 28 and 29 show as a gray-scale gradient the analysis time for both methods as the scope grows.

Figs. 28 and 29 show the relation between a tighter bound and the analysis time. It is easy to see that tightening the bound contributes in allowing the analysis to finish within the time limit for larger scopes.

5.4 Analysis of Bug-Free Code

In this section we present the results of comparing TACO with tight bounds with JForge, another SAT-based tool for Java code analysis. The results are shown in Table 4.

Table 4 shows that as the scope grows, in most cases (as the cell highlighting shows) TACO requires a smaller amount of time than JForge. While we will not present a detailed analysis of memory consumption, it is our experience that TACO uses less memory than JForge, both during translation to a propositional formula and during

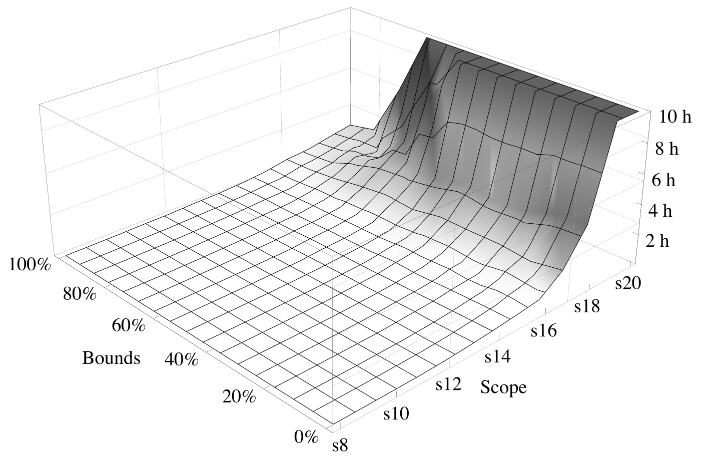


Fig. 29. Analysis time for method *insert* of CList as scope and bound tightness grows.

#Node			5	7	10	12	15	17	20
LList	contains	JF	00:01	02:00	TO	TO	TO	TO	TO
		T	00:03	00:04	00:05	00:06	00:07	00:09	00:15
	insert	JF	00:02	04:56	TO	TO	TO	TO	TO
		T	00:04	00:05	00:07	00:08	00:13	00:26	00:40
	remove	JF	00:04	21:51	TO	TO	TO	TO	TO
		T	00:04	00:06	00:11	00:12	00:17	00:33	00:42
AList	contains	JF	00:02	05:01	TO	TO	TO	TO	TO
		T	00:04	00:06	00:16	00:22	00:27	00:58	02:49
	insert	JF	00:03	11:52	TO	TO	TO	TO	TO
		T	00:04	00:05	00:07	00:08	00:12	00:16	00:25
	remove	JF	00:18	73:27	TO	TO	TO	TO	TO
		T	00:05	00:06	00:17	00:31	01:08	03:13	08:24
CList	contains	JF	00:05	10:23	TO	TO	TO	TO	TO
		T	00:11	00:19	01:23	01:56	05:51	07:25	06:54
	insert	JF	00:20	201:54	TO	TO	TO	TO	TO
		T	00:09	00:12	00:16	00:28	01:07	02:01	04:57
	remove	JF	02:28	TO	TO	TO	TO	TO	TO
		T	00:27	00:59	03:26	03:43	28:18	57:23	89:17
BSTree	contains	JF	09:41	TO	TO	TO	TO	TO	TO
		T	00:01	00:25	114:06	TO	TO	TO	TO
	insert	JF	03:46	TO	TO	TO	TO	TO	TO
		T	00:01	00:26	32:58	TO	TO	TO	TO
	remove	JF	OofM	OofM	OofM	OofM	OofM	OofM	OofM
		T	08:19	102:46	TO	TO	TO	TO	TO
TreeSet	find	JF	00:42	117:49	TO	TO	TO	TO	TO
		T	00:04	00:10	01:56	12:43	58:54	305:06	TO
	insert	JF	OofM	OofM	OofM	OofM	OofM	OofM	OofM
		T	00:43	08:44	TO	TO	TO	TO	TO
	remove	JF	OofM	OofM	OofM	OofM	OofM	OofM	OofM
		T	00:53	06:25	196:58	TO	TO	TO	TO
AVL	find	JF	00:26	190:10	TO	TO	TO	TO	TO
		T	00:03	00:06	00:36	01:41	08:20	33:06	179:53
	findMax	JF	00:06	49:49	TO	TO	TO	TO	TO
		T	00:01	00:01	00:03	00:04	00:09	00:13	01:09
	insert	JF	OofM	OofM	OofM	OofM	OofM	OofM	OofM
		T	00:07	00:34	04:47	21:53	173:57	TO	TO
	remove	JF	OofM	OofM	OofM	OofM	OofM	OofM	OofM
		T	00:10	01:25	06:36	60:17	128:47	TO	TO
BHeap	findMin	JF	00:22	83:07	TO	TO	TO	TO	TO
		T	00:05	00:08	00:14	00:17	01:31	02:51	07:26
	decKey	JF	01:48	TO	TO	TO	TO	TO	TO
		T	00:16	01:13	30:26	TO	TO	TO	TO
	insert	JF	73:47	TO	TO	TO	TO	TO	TO
		T	01:54	08:08	37:30	218:13	TO	TO	TO

TABLE 4
Comparison of code analysis times for 10 loop unrolls using JForge (JF) and TACO (T).

SAT-solving. The analysis time using TACO reported in Table 4 does not include the cost of computing bounds (the time spent in discovering tighter bounds was given in Table 3). Still, adding these times does not yield a TO for any of the analyses that did not exceed 10 hours.

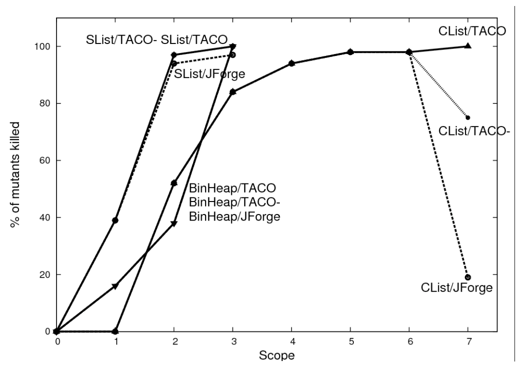


Fig. 30. Efficacy of JForge, TACO$^-$ and TACO for mutants killing.

5.5 Bug Detection Using TACO

In this section we report on our experiments using TACO in order to detect faults, and will compare TACO to other tools. We will analyze 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 TACO with ESC/Java2 [9], JavaPathFinder [44], and Sireum/Kiasan [13].

5.5.1 Detecting Mutants

In order to compare JForge, TACO$^-$ and TACO we will generate mutants for the chosen methods using the muJava [37] mutant generator tool. After manually removing from the mutants set those mutants that either were equivalent to the original methods or that 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 to 1 hour.

In Fig. 30 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 strictly kill more mutants than TACO$^-$ and JForge in the CList example. We can also see that as the scope grows, 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 $T = \sum_{1 \leq i \leq k} T_i$.

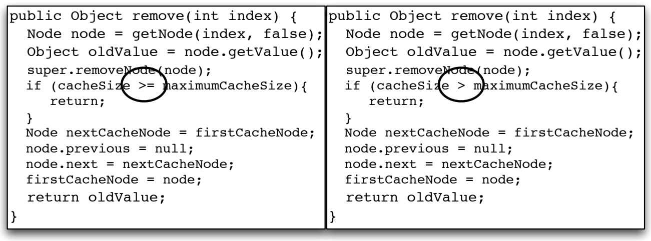
Notice first that the previous procedure favors TACO$^-$ and JForge. In effect, if a tool is used in isolation we cannot set an accurate scope limit beforehand (it is the user's responsibility to set the limit). If a scope smaller than the necessary one is chosen, then killable mutants will survive. 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.

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 5, 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 5
Analysis times for mutant 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 6 we present analysis times using all the tools. Table 6 shows that TACO, Java PathFinder and Kiasan were the only tools that succeeded in killing all the mutants. Since the fragment of JML supported by ESC/Java2 is not expressive enough to model the invariant from class BHeap, we did not run that experiment.



```
public Object remove(int index) {
  Node node = getNode(index, false);
  Object oldValue = node.getValue();
  super.removeNode(node);
  if (cacheSize >= maximumCacheSize){
     return;
  }
  Node nextCacheNode = firstCacheNode;
  node.previous = null;
  node.next = nextCacheNode;
  firstCacheNode = node;
  return oldValue;
}
```

(a)

```
public Object remove(int index) {
  Node node = getNode(index, false);
  Object oldValue = node.getValue();
  super.removeNode(node);
  if (cacheSize > maximumCacheSize){
     return;
  }
  Node nextCacheNode = firstCacheNode;
  node.previous = null;
  node.next = nextCacheNode;
  firstCacheNode = node;
  return oldValue;
}
```

(b)

Fig. 31. Code snippets from `CList.remove` (a), and a bug-seeded version (b).

	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 6
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.

5.5.2 *Detecting a Seeded Non-Trivial Bug*

Notice that in the previous section, although we chose the supposedly most complex mutants, these are still simple in the sense that they can be killed using small scopes. In this section we are interested in studying the performance of these tools in a context where a larger amount of nodes are needed to find a violation of the specification. In this sense, we focus on the linked data structure for class `CList`. This data structure is composed by the actual (circular) list, and a singly linked list (the cache). The cache list has a maximum size, "`maximumCacheSize`" (`maxCS`), set in the actual code to a default value of 20 nodes. When a node is removed from the circular list, it is added to the cache (unless the cache is full). Let us consider the code snippet from `remove` presented in Fig. 31.(a). Fig. 31.(b) gives us a bug-seeded version. A failure occurs in the bug-seeded code when a node is removed and the cache is full. In effect, if the maximum cache size is set to the default of 20, a 21st element can be added to the cache. This leads to a violation of the invariant that constrains the cache size to be at most the value of the maximum cache size field.

In Table 7 we report analysis information after looking for the bug in the bug-seeded code (BS), for varying numbers of loop unrolls in method `super.removeNode`. We have tailored the bug-seeded code (and its contract), to be analyzed using the same tool set we have applied in the previous section for analyzing the more complex mutants.

We computed a bound for TACO in 27:04 using one iteration of the iterative algorithm of Fig. 20. Table 7 shows that many times it is not necessary to compute the tightest bound, but rather thin the initial bound with a few iterations of the algorithm in order to achieve a significant speedup in analysis time. The debugging process consists of running a tool (such as TACO, JForge, etc.) and, if a bug is found, correcting the error and starting over to look for further bugs. Unlike JForge (where each analysis is independent of the previous ones), the same bound can be used by TACO for looking for all the bugs in the code. Therefore, the time required for computing the bound can be amortized among these bugs. Since the bound does not depend on the number of unrolls, in Table 7 we have divided 27:04 among the 7 experiments, adding 03:52 to each experiment. Time is reported as "bound computation time" + "SAT-solving time."

LU	JForge	ESC/Java2	JPF	Kiasan	TACO
4	OofM(227)	OofM(206)	TO	OofM(4)	03:52 + 03:56
6	TO	OofM(207)	TO	OofM(4)	03:52 + 31:14
8	OofM(287)	OofM(213)	TO	OofM(4)	03:52 + 33:23
10	05:40:22	OofM(215)	TO	OofM(4)	03:52 + 00:11
12	06:53:04	OofM(219)	TO	OofM(4)	03:52 + 03:30
15	24:08	OofM(219)	TO	OofM(4)	03:52 + 15:00
20	TO	OofM(218)	TO	OofM(4)	03:52 + 00:06

TABLE 7
Outcome of the analysis `maxCS` = 20. Ten hours timeout.

We also compared with Boogie [3] using Z3 [16] as the back-end SMT solver. In order to produce Boogie code we used Dafny [35] as the high-level programming and specification language. When ran on the bug-seeded code with 10 loop unrolls, Boogie produced in the order of 50 warning messages signaling potential bugs. A careful inspection allowed us to conclude that all warnings produced by Boogie were false warnings.

Since most tools failed to find the bug with `maxCS` = 20, we also considered a version of the code with up to 2 loop unrolls and varying values for `maxCS`; in this way the bug can be found in smaller heaps. Table 8 reports the corresponding analysis times. In TACO we have restricted the algorithm that computes the bound for each scope to run for 30 minutes at most.

The code has a fault that requires building a non-trivial heap to expose it. The technique introduced in this article made TACO the *only* tool capable of finding the bug in *all* cases reported in Tables 7 and 8. When the size of the code is small (2 loop unrolls in Table 8), tools based on model checking were able to find the bug. They failed on larger code, which shows that in the example TACO scales better. Tools based on SMT solving systematically failed to expose the seeded bug.

mCS	JForge	ESC/Java2	JPF	Kiasan	TACO
5	00:13	OofM(187)	00:07	00:18	01:21 + 00:01
10	05:13	OofM(212)	00:20	00:43	02:25 + 00:11
13	OofM(529)	OofM(221)	00:38	OofM(3)	05:27 + 00:32
15	OofM(334)	OofM(214)	00:53	OofM(3)	21:31 + 00:15
18	14:04	OofM(200)	01:27	OofM(4)	30:00 + 02:27
20	OofM(494)	OofM(556)	02:17	OofM(4)	30:00 + 02:11

TABLE 8
Up to 2 unrolls and varying `maxCS`. 10 hours timeout.

5.5.3 Detecting a Previously Unknown Fault

As we mentioned in [26], TACO found a previously unreported bug in method `ExtractMin` of class `BHeap`. A distinguishing characteristic of this fault is that it cannot be reproduced using mutation because the smallest input that produces a failure has 13 nodes, and as we showed before in Sec. 5.5.1, all mutants were killed with only 3 nodes. Another interesting attribute of this defect is that it is not easily identified as a bug introduced as a programmer typo. What is more, the fault is not trivially discovered by team revision.

The input datum leading to the failure is presented in Fig. 32. Notice that at least 4 loop unrolls were required in TACO in order to exhibit the failure. In Table 9 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 9
Analysis of a non-trivial bug.

5.6 Threats to Validity

We begin by discussing how representative the selected case studies are. As discussed in [45], container classes have become ubiquitous. Therefore, providing confidence about their correctness is an important task in itself. But, as argued in [41], 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 program analysis community (see for instance [6], [15], [30], [45]).

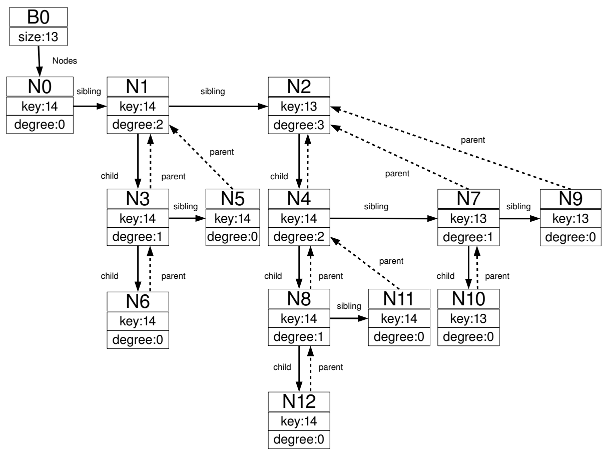


Fig. 32. A 13 nodes heap that exhibits the failure in method `ExtractMin`.

Despite the proof of correctness presented in Sect. 3.3, one might be concerned about the way in which the generation of the symmetry breaking predicates was implemented. In order to validate our prototype we checked the number of non-isomorphic instances explored by TACO, against Korat [6]. For each class in the benchmark the number of valid instances (up to 10 nodes in the heap) matched. Table 10 contains the number of generated instances by both tools for instances having from 3 to 10 nodes. In all cases a timeout of 2 hours was set.

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 5.4 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. It is a stress test that necessarily arises during actual bug finding.

In Section 5.5 we compare 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.

6 Related Work

In Sec. 3 we analyzed related work on heap canonicalization. In Sec. 5 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 [32]. It allows the annotation of Java-like code using Alloy as the annotation language. The translation proposed in [32] does not differ in major ways from the one we implement. Analysis using AAL does not include any computation of bounds for fields.

In [43] 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.

Thesis [31] presents the foundations of TestEra [34], and shows an attempt at automatically eliminating symmetries from a Java heap. Only heaps with a singly-rooted acyclic backbone are considered, which requires the user to actually identify the (acyclic) backbone. For instance, class AList in the benchmark, describing circular, doubly-linked lists, does not possess an acyclic backbone and TestEra (unlike TACO) will require the user to provide symmetry breaking predicates. Although no technical details are given, the mention of the use of the total-ordering module provided by Alloy makes the approach unsuitable in the context of TACO.

Saturn [47] 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.

	#Nodes	3	4	5	6	7	8	9	10
LList	#instances	4	5	6	7	8	9	10	11
	Korat (secs)	0.13	0.13	0.13	0.13	0.14	0.13	0.13	0.13
	TACO (secs)	0.35	0.36	0.42	0.45	0.49	0.55	0.62	0.8
AList	#instances	3	4	5	6	7	8	9	10
	Korat (secs)	0.14	0.13	0.14	0.14	0.14	0.15	0.14	0.15
	TACO (secs)	0.53	0.59	0.61	0.73	0.88	0.94	1.11	1.56
CList	#instances	6	10	15	21	28	36	45	55
	Korat (secs)	0.15	0.16	0.15	0.16	0.16	0.17	0.18	0.19
	TACO (secs)	0.64	0.81	0.98	1.12	1.35	1.63	2.13	2.7
BSTree	#instances	9	23	65	197	626	2056	6918	23714
	Korat (secs)	0.14	0.15	0.18	0.25	0.33	0.39	0.58	1.14
	TACO (secs)	0.4	0.52	0.64	0.97	2.2	2.88	5.79	16.04
TreeSet	#instances	6	10	18	34	67	123	213	377
	Korat (secs)	0.20	0.26	0.28	0.38	0.52	0.70	1.55	5.76
	TACO (secs)	0.55	0.61	0.78	1.07	1.73	2.08	2.66	3.47
AVL	#instances	5	9	15	16	36	68	112	172
	Korat (secs)	0.18	0.27	0.37	1.61	29.26	795.67	TO	TO
	TACO (secs)	0.49	0.59	0.64	0.77	0.93	1.46	1.86	2.19
BHeap	#instances	4	5	6	7	8	9	10	11
	Korat (secs)	0.23	0.26	0.36	0.43	0.55	0.96	2.84	11.56
	TACO (secs)	0.63	0.67	0.93	1.16	1.47	1.70	2.19	2.85

TABLE 10
Number of instances generated and time consumed by TACO and Korat considering from 3 to 10 nodes.

Saturn can check assertions written as C "assert" statements. Its assertion language is not as declarative as our extension of JML.

VCC [7] targets concurrent C code, and uses SMT solving as the underlying technology.

F-Soft [28] also analyzes 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 [40]. Our technique produces tighter upper bounds because it does not compute feasible intervals for variables, but instead checks each individual value.

Calysto [2] performs an inter procedural analysis based on symbolic execution. TACO is evaluated on single methods as a means to assess its scalability at the intraprocedural level.

Jahob [5] allows the unbounded verification of complex properties over linked data structures (such as binary trees, red black trees, etc). As Jahob's language was designed as a proof language, it provides language constructs for identifying lemmas, witnesses of existential quantifications, patterns for instantiating universal quantifiers, proofs by induction, etc. Although the expressiveness of this proof language allows the user to write very useful annotations for the underlying decision procedures (which allows the verification of very complex properties), it is easy to see that the annotation process goes far beyond the specification of a program's behavior.

jStar [18] is an automatic tool for modular verification of sequential Java programs. It is based on the abstraction techniques for *shape analysis* developed in [17]. The user provides specifications in the form of pre/post conditions, while loop invariants are automatically synthesized. Since jStar over-approximates the program behavior, the given verdict does not depend on a user-provided scope of analysis. Like in the case of Jahob, a jStar user must add additional annotations beyond those specifying program behavior. In particular, she or he must provide:

- a logical theory (used by the theorem prover for deciding entailment and other kinds of implications), and
- an abstraction function (used to ensure convergence in the fixed-point computation of loop invariants).

In our experience, by solely providing the program's behavior specification, neither Jahob nor jStar succeeded in verifying the provided specifications, nor provided understandable counterexamples.

Unlike Jahob or jStar, TACO does not require user-provided rules apart from the JML annotations.

7 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, while this article presents an approach to bound computation newer than the one presented in [26], we are working further, more efficient methods for distributed bound computation.

We are developing a prototype tool that, using dataflow analysis [8], propagates the tight bounds computed for the relational variables representing the initial state, and generates bounds for subsequent states.

We have obtained encouraging results on parallel analysis of code by conveniently splitting tight bounds into tighter bounds.

None of the container classes presented in Sect. 5 possess a complex class hierarchy. More experiments are required to assess the performance of our approach under such circumstances.

The techniques presented in the article are quite general. We plan to test the effect of these techniques on related tools. Explicit state model checkers (such as Java Pathfinder) can use tight bounds in order to prune the state space when a state contains edges that lay outside the bound. Korat [6] can avoid evaluating the `repOk` method whenever the state is not contained in the bounds. Running a simple membership test will often be less expensive than running a `repOk` method. Tools that are similar to TACO (such as Miniatur and JForge), can make direct use of the presented techniques. Similarly, Squander [39], a tool for execution of Alloy-like Java specifications, could profit from applying both the specialized symmetry breaking and the propositional variables reduction.

8 Acknowledgements

We thank Elena Morin for proofreading this article. We wish to thank all the reviewers for their comments and suggestions. We thank Greg Dennis and Kuat Yessenov for their timely help on using JForge and JMLForge. We thank Robby for answering our questions about Sireum/Kiasan. We also thank Esteban Mocskos for his support on using the CeCAR cluster.

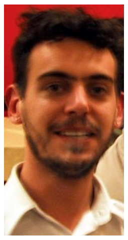
References

[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] Babić D. and Hu A. J., *Calysto: Scalable and Precise Extended Static Checking*, in Proceedings of ICSE 2008.
[3] Barnett M., Chang B.E, DeLine R., Jacobs B., Leino K.R.M. *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. FMCO 2005: pp. 364–387.
[4] 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.
[5] 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.
[6] Boyapati C., Khurshid S., Marinov D., *Korat: automated testing based on Java predicates*, in ISSTA 2002, pp. 123–133.
[7] Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W. and Tobies S., *VCC: A Practical System for Verifying Concurrent C*, in Proceedings of TPHOLs 2009.
[8] Cousot P., Cousot R. *Systematic Design of Program Analysis Frameworks*, in POPL 1979: pp. 269–282
[9] 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.
[10] Clarke E., Kroening D., Lerda F., *A Tool for Checking ANSI-C Programs*, in TACAS 2004, LNCS 2988, pp. 168–176.
[11] Cook, S., *The Complexity of Theorem-Proving Procedures*, in Proceedings of the Third Annual ACM Symposium on Theory of Computing, pp. 151–158, ACM, 1971.
[12] 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.
[13] 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.
[14] Dennis, G, *A Relational Framework for Bounded Program Verification*. PhD thesis. MIT Press, September 2009.
[15] Dennis, G., Chang, F., Jackson, D., *Modular Verification of Code with SAT*. in ISSTA'06, pp. 109–120, 2006.
[16] Mendonça de Moura L., Bjørner N. *Z3: An Efficient SMT Solver*. TACAS 2008, pp. 337–340.
[17] Distefano D., O'Hearn P., Yang H., *A local shape analysis based on separation logic*, in 12th TACAS, LNCS 3920, pp287-302, 2006.
[18] Distefano D., Parkinson M., *jStar: Towards Practical Verification for Java*, in OOPSLA 2008, pp. 213-226, ACM.
[19] Dennis, G., Yessenov, K., Jackson D., *Bounded Verification of Voting Software*. in VSTTE 2008. Toronto, Canada, October 2008.
[20] Dolby J., Vaziri M., Tip F., *Finding Bugs Efficiently with a SAT Solver*, in ESEC/FSE'07, pp. 195–204, ACM Press, 2007.
[21] Edwards J., Jackson D., Torlak E., Yeung V., *Subtypes for constraint decomposition*. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA04), Boston, MA, July 2004.
[22] Flanagan, C., Leino, R., Lillibridge, M., Nelson, G., Saxe, J., Stata, R., *Extended static checking for Java*, In PLDI 2002, pp. 234–245.
[23] Frias, M. F., Galeotti, J. P., Lopez Pombo, C. G., Aguirre, N., *DynAlloy: Upgrading Alloy with Actions*, in ICSE'05, pp. 442–450, 2005.
[24] Frias, M. F., Lopez Pombo, C. G., Galeotti, J. P., Aguirre, N., *Efficient Analysis of DynAlloy Specifications*, in ACM-TOSEM, Vol. 17(1), 2007.
[25] 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.
[26] Galeotti, J. P., Rosner, N., López Pombo, C. G., Frias, M. F., *Analysis of Invariants for Efficient Bounded Verification*, in Proceedings of ISSTA 2010, pp. 25–36, 2010.
[27] Iosif R., *Symmetry Reduction Criteria for Software Model Checking*. SPIN 2002: 22-41
[28] 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.
[29] Jackson, D., *Software Abstractions*. MIT Press, 2006.
[30] Jackson, D., Vaziri, M., *Finding bugs with a constraint solver*, in ISSTA'00, pp. 14-25, 2000.
[31] Khurshid, S., *Generating Structurally Complex Tests from Declarative Constraints*. MIT PhD Thesis. Feb 2003. http://sdg.csail.mit.edu/pubs/theses/khurshid.phd.pdf
[32] Khurshid, S., Marinov, D., Jackson, D., *An analyzable annotation language*. In OOPSLA 2002, pp. 231-245.
[33] Khurshid, S., Marinov, D., Shlyakhter, I., Jackson, D., *A Case for Efficient Solution Enumeration*, in SAT 2003, LNCS 2919, pp. 272–286.
[34] Khurshid, S., Marinov, D., *TestEra: Specification-Based Testing of Java Programs Using SAT*, Automated Software Engineering Journal vol. 11(4), pp. 403–434, 2004.
[35] Leino K.R.M., *Specification and verification of Object-Oriented Software*, Lecture Notes from Marktoberdorf International Summer School 2008.
[36] Harel D., Kozen D., and Tiuryn J. *Dynamic logic. Foundations of Computing*. MIT Press, 2000
[37] 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.
[38] Musuvathi M., Dill, D. L., *An Incremental Heap Canonicalization Algorithm*, in SPIN 2005: 28-42
[39] Milicevic A., Rayside D., Yessenov K., Jackson D., *Unifying Execution of Imperative and Declarative Code*. In 33rd International Conference on Software Engineering (ICSE), Waikiki, Honolulu, Hawaii, May 2011.
[40] Rugina, R., Rinard, M. C., *Symbolic bounds analysis of pointers, array indices, and accessed memory regions*, in PLDI 2000, pp. 182–195, 2000.
[41] Siddiqui, J. H., Khurshid, S., *An Empirical Study of Structural Constraint Solving Techniques*, in ICFEM 2009, LNCS 5885, 88–106, 2009.

[42] Torlak E., Jackson, D., *Kodkod: A Relational Model Finder*. in TACAS '07, LNCS 4425, pp. 632–647.
[43] Vaziri, M., Jackson, D., *Checking Properties of Heap-Manipulating Procedures with a Constraint Solver*, in TACAS 2003, pp. 505-520.
[44] Visser W., Havelund K., Brat G., Park S. and Lerda F., *Model Checking Programs*, ASE Journal, Vol.10, N.2, 2003.
[45] Visser W., Păsăreanu C. S., Pelánek R., *Test Input Generation for Java Containers using State Matching*, in ISSTA 2006, pp. 37–48, 2006.
[46] Visser W., *Private communication*, February 2nd., 2010.
[47] Xie, Y., Aiken, A., *Saturn: A scalable framework for error detection using Boolean satisfiability*. in ACM TOPLAS, 29(3): (2007).

Juan P. Galeotti is a post-doc researcher at Saarland University, Saarbrücken, Germany. His research interests include software verification, program analysis, automatic test case generation and programming languages design. He has been awarded with the José A. Estenssoro doctoral grant from YPF Foundation.

Nicolás Rosner is a doctoral candidate in computer science at the Department of Computer Science, FCEyN, Universidad de Buenos Aires, where he is also a teaching assistant. His research interests include relational model finding, cluster computing and distributed SAT-solving.



Carlos G. López Pombo is an Assistant Professor at the Department of Computer Science, FCEyN, Universidad de Buenos Aires, and an Assistant Researcher at CONICET. His research centers on formal methods for software verification and validation, and is currently working on formal foundations for component-based software design and on formal frameworks supporting service oriented architecture. His research interests also include the development of automatic and interactive tools for software verification.

Marcelo F. Frias is a Professor of Computer Science at the Buenos Aires Institute of Technology. His interests range from formal logic and universal algebra, to relational methods and their application for (semi-)automated software validation and verification. He is a member of IFIP Working Group 2.2.