Distributed SAT-Based Computation of Relational Tight Bounds

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

Department of Computer Science, FCEyN, Universidad de Buenos Aires
e-mail: {jgaleotti, nrosner, clpombo, mfrias}@dc.uba.ar

Abstract. SAT based analysis of code consists on providing an appropriate translation of code to a SAT problem, and using a SAT solver to analyze properties of the source program. This process can be improved if bounds for relations modeling class fields are introduced ala KodKod. In this article we present a distributed algorithm for automated inference of tight field bounds. From a technical point of view, the algorithm relies on a novel symmetry breaking predicate for program heaps. We present experimental results that support our claims.

1 Introduction

SAT-solving has been widely used as a tool for automated analysis of code. The general approach consists on translating source code and some assertion to be analyzed to a SAT-problem (a propositional formula), and then use an off-the-shelf SAT solver to look for a satisfying valuation of the propositional variables. If such valuation is found, it can be transformed into a valid execution of the source code that falsifies the given assertion. While directly translating code to a propositional formula is a valid way to proceed, our approach consists on translating the source code to an Alloy model [4] using the DynJAlloy translator [3]. Alloy is a modeling language that includes several constructs ubiquitous in object-orientation. An interesting feature of the current version of Alloy (Alloy 4) is that it translates Alloy models to KodKod [8] models. KodKod, the relational engine of Alloy includes, as a distinguishing feature, the possibility of introducing bounds for relations occurring in a model. This allows to remove some propositional variables in the translation of an Alloy mode to a propositional formula, which simplifies the SAT-solving procedure.

The contribution of this article is a fully automated technique for computing tight bounds in an Alloy model obtained as the result of the translation of a piece of JML-annotated code. In order to compute the bounds we introduce a novel symmetry breaking predicate that is tailored for the description of heaps in Alloy. Since for many data structures the heap must have certain topological properties (think for instance of singly linked lists or binary search trees), it is possible, thanks to the reduction of symmetries, to determine that certain edges between nodes cannot occur in the data structure (and can therefore be removed from the bound).

It is well known that less propositional variables in the propositional formula mean a more analyzable SAT-problem. Therefore, computing tight relational bounds will result in easier to perform code analyses.

The article is organized as follows. In Section 2 we present a brief introduction to Alloy, KodKod and DynJAlloy. In Section 3 we introduce the symmetry breaking predicate and prove some desirable properties. In Section 4 we present our technique for automated computation of tight relational bounds. In Section 6 we present experimental results showing that the technique presented in Section 4 is effective. In Section 7 we compare with the most related work to ours. Finally, in Section 8 we preset our conclusions and some lines for further work.

2 On Alloy and KodKod and DynJAlloy

We will present Alloy by means of an example. In Fig. 1 we present an Alloy model for singly linked lists. An Alloy model consists of definitions of data types (called *signatures* in Alloy notation), which, like classes in object-oriented programming languages, may include fields. Signatures denote sets of atoms. A field (for instance *head* in Fig. 1) denotes a relation from atoms of the class the field belongs to, to elements in the codomain. For example, *head* is a functional relation from List atoms to LNode atoms or the null atom. The modifier *one* before a signature definition means that the domain corresponding to the signature will have a single atom. In order to abbreviate and modularize notation it is possible to define function and predicates. Functions compute relational expressions, while predicates evaluate a relational formula. Expressions are built from relations. Signatures are unary relations, and fields *head* or *value* are binary relations ($head \subseteq List \times (LNode + null)$). It is possible to have relations of higher arity because fields may have relations in their codomain. Alloy operations include all natural operations on relations. For instance union ($+$), intersection ($\&\&$), difference ($-$), composition ($.$), reflexive-transitive closure ($*$) and transitive closure ($\hat{}$). Besides these operations, whose use is shown in Fig. 1, there is also the (unary) transposition operation, denoted $\sim$. Besides, there are also constant relations such as *iden* (which stands for the binary identity relation) and *univ* (the set of all atoms). For a detailed description of Alloy, see [4].

Alloy's relational kernel is called KodKod. While in Alloy 3 models were directly translated to propositional formulas, in Alloy 4 models are translated to KodKod models. These models are then translated to SAT problems. A major characteristic of KodKod is the possibility of defining lower and upper bounds for relations (relations such as the fields *value* or *next*). If a pair is included in the lower bound, it must be part of any possible semantic interpretation for that relation. Therefore, the propositional variable representing that pair can be substituted by the truth value *true*. Similarly, if a pair is not included in the upper bound, we know that none of the interpretations of the corresponding relation may have that pair. Therefore, the propositional variable representing that pair can be substituted by the truth value *false*.

```
sig List {
   head : LNode + null
}

one sig null {}

sig LNode {
   value : Int,
   next : LNode + null
}

fun reachableLNodes[l : List] : set LNode {
   l.head.*next - null
}

pred noRepetitions[l : List]{
   all n1, n2 : reachableLNodes[l] |
      n1.value = n2.value implies n1 = n2
}

fact acyclic { all l : List, n : reachableLNodes[l] | n !in n.^next }

assert noRepetitionsTrue { all l : List | noRepetitions[l] }

check noRepetitionsTrue for 3 but exactly 1 List, 10 LNode, 5 int
```

Fig. 1. An Alloy model for singly linked lists.

DynJAlloy [2] is our tool for SAT-based Java code analysis. It translates JML-annotated Java code [6] into DynAlloy, and the resulting DynAlloy model into an Alloy 4 model. The Alloy Analyzer [4] is then used in order to look for counterexamples of the JML annotation.

3 A New Predicate for Symmetry Breaking in Heaps

In this section we present a novel predicate for symmetry breaking in heaps. Let us consider the model for singly linked lists presented in Fig. 1. The following predicate

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

describes acyclic lists. Running predicate acyclic using the command

```
run acyclic for exactly 4 Object,
                exactly 1 List,
                exactly 3 LNode,
                exactly 3 Data
```

yields the model from Fig. 2. Another run may return the model from Fig. 3.

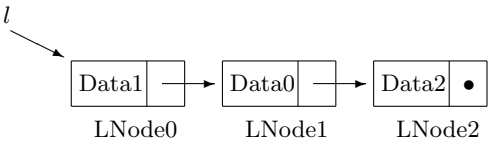

Fig. 2. An acyclic list.

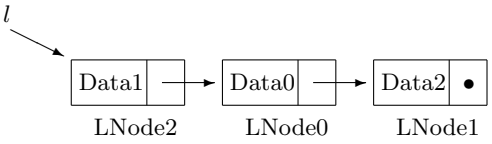

Fig. 3. Another acyclic list, equivalent to the one in Fig. 2.

Notice that the list in Fig. 3 is equivalent to the one in Fig. 2. Both serve as "runs" for predicate "acyclic". At the same time, cyclic lists are not models. And there are many cyclic lists which are essentially the same up-to permutations of signature LNode. Pruning the state space by removing permutations on signature LNode prevents the SAT-solver from trying structures that cannot lead to a valid model. For singly linked lists, a predicate forcing nodes to be used in the order LNode0 → LNode1 → LNode2 → ... removes symmetries. Unfortunately, it will not work, for instance, with cyclic structures.

In the remaining part of this section we will present a symmetry breaking predicate that lists nodes in a rooted heap in breadth-first order. Moreover, the predicate will work even in the presence of cyclic structures. In order to simplify the presentation, we will do it through an example. Let us consider the Alloy model for binary trees with information in the nodes presented in Fig. 4.

```
sig Tree {
   root : (TNode + null)
}

sig TNode {
   left : (TNode + null),
   value : Data,
   right : (TNode + null)
}
```

Fig. 4. A model for binary trees.

Notice that field root provides a handle on the reachable part of the heap. This is what we will call a rooted heap. Definition 2 formalizes this notion.

Definition 1. *Given a class C, a field $f : C \to D$ is called* recursive *if $D \cap C \neq \emptyset$. For instance, fields left and right are recursive in class TNode.*

Definition 2. *Given a class C with recursive fields $f_i : C \to D_i$ $(1 \leq i \leq k)$, a rooted C-heap is a graph $\langle N, E, L, R \rangle$, where:*

1. *$N \subseteq C \cup \bigcup_{1 \leq i \leq k} D_i$.*
2. *Edges from E are labeled with symbols from the set $L = \{f_1, \ldots, f_k\}$.*
3. *$R \in N$ is such that $N \subseteq R.*(f_1 + \cdots + f_k)$.*

In order to avoid symmetries we will instrument Alloy models resulting from the translation of code by doing the following:

1. If the scope for signature C is k, we include singletons $C_0, \ldots, C_{k-1}$:

   ```
   one sig TNode0,...,TNodek-1 extends TNode {}
   ```

2. Each recursive field `r : C + D` from C is split into two fields `fr : lone (C + D)` and `br : lone (C + D')` (D' differs from D in that the `null` value is not allowed in D'). In our example the resulting fields are:
   - `fleft : lone (TNode + null)`,
   - `bleft : lone TNode`,
   - `fright : lone (TNode + null)`, and
   - `bright : lone TNode`.
3. Facts forcing the SAT-solver to choose nodes in a specific order.

The new fields obtained (that substitute the original ones) are meant to split the behavior of the original fields between "forward" arcs and "backwards" arcs. Forward arcs map nodes to greater nodes ($TNode_i \to TNode_j$, with $i < j$), while backwards 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.

Since the original fields are all total functions, we need to add new facts stating that for each recursive field r_i, the domains of fr_i and br_i form a partition. Therefore, $fr_i + br_i$ is a well defined total function. For our example we have:

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

The facts inducing the ordering use auxiliary functions defined as follows. Function *next* establishes a linear order between nodes (TNode1 < TNode2 $< \cdots <$ TNodek). Function *min* returns the least node in an input set according to the *next* ordering (notice that if the input set is empty, so is the output). Function *prevs* returns the nodes smaller than the input parameter.

```
 – fun next[] : TNode -> lone TNode {
      TNode0->TNode1 + TNode1->TNode2 + ... + TNodek-2->TNodek-1
   }
 – fun min [ns: set TNode] : lone TNode { ns - ns.^(next[]) }
 – fun prevs[n : TNode] : set TNode { n.^(~next[]) }
```

The following facts induce the appropriate ordering on the way nodes are chosen by the SAT-solver.

```
fact rootIsTheFirstNode { Tree.root in TNode0+null }

fact parentSmallerThanChildren {
   all n : Tree.root.*(fleft + fright) - null |
      min[fleft.n] in prevs[n] and min[fright.n] in prevs[n]
}

fact howToOrderTwoNodesWithDifferentParents {
   all disj n1, n2 : Tree.root.*(fleft + fright) - null |
      ( some (fleft.n1 + fright.n1) and
        some (fleft.n2 + fright.n2) and
        min[fleft.n1 + fright.n1] in prevs[min[fleft.n2 + fright.n2]]
      ) implies n1 in prevs[n2]
}

fact howToOrderTwoNodesWithSameParent {
   all disj n1, n2 : Tree.root.*(fleft + fright) - null |
   let a = min[fleft.n1 + fright.n1] |
   let b = min[fleft.n2 + fright.n2] |
   (some (fleft.n1 + fright.n1) and a = b and a.fleft = n1 and a.fright = n2)
      implies n2 = n1.next[]
}

fact initialBounds {
   fleft  in TNode0 -> (TNode1 + ... + TNodek + null) +
             TNode1 -> (TNode2 + ... + TNodek + null) +
             ...
             TNodek -> null
   and
   fright in TNode0 -> (TNode1 + ... + TNodek + null) +
             TNode1 -> (TNode2 + ... + TNodek + null) +
             ...
             TNodek -> null
   and
   bleft  in TNode0 -> TNode0 +
             TNode1 -> (TNode1 + TNode0) +
             ...
```

```
            TNodek -> (TNodek + ... + TNode0)
   and
   bright in TNode0 -> TNode0 +
            TNode1 -> (TNode1 + TNode0) +
            ...
            TNodek -> (TNodek + ... + TNode0)
}

fact prefixComplete {
   all n : Tree.root.*(fleft + fright) - null |
      prevs[n] in Tree.root.*(fleft + fright)
}
```

Since a node may have multiple parents through fields $f_1, \ldots, f_k$, in the breadth-first listing of the nodes we will consider the smallest (according to the ordering on nodes) parent according to all the fields $f_1, \ldots, f_k$ as the one that determines the order of the node. We will call that parent the $(f_1 + \cdots + f_k)$-*min-parent*. The axioms are explained in English as follows

parentSmallerThanChildren: Every node is larger than its f-min-parent for each recursive forward field f.

howToOrderTwoNodesWithDifferentParents: If n_1 and n_2 are distinct nodes and the $(f_1 + \cdots + f_k)$-min-parent of n_1 is less than the $(f_1 + \cdots + f_k)$-min-parent of n_2, then n_1 is smaller that n_2.

howToOrderTwoNodesWithSameParent: In case nodes n_1 and n_2 have the same $(f_1 + \cdots + f_k)$-min-parent, then the symmetry is broken by listing first the one pointed to by field f_1, second the one pointed to by field f_2, and so on.

initialBounds: Forward fields point to greater nodes or to null. Backwards fields point nodes to smaller or equal nodes.

prefixComplete: If a node is in the graph, all smaller nodes are also in the graph.

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

```
fact acyclic { all n : Tree.root.*(left + right) - null |
   n !in n.^(left + right) }
```

in the instrumented model it is replaced by the fact

```
fact acyclic { all n :
   Tree.root.*(fleft + bleft + fright + bright) - null |
      n !in n.^(fleft + bleft + fright + bright) }
```

Given a rooted C-heap $G = \langle N, E, L, R \rangle$, it is possible to define a new rooted C-heap G' whose edges are labeled with forward and backwards recursive fields. Moreover, G' satisfies:

1. G is a model if and only if G' is a model.
2. The nodes in G' are chosen in breadth-first order.

Theorem 1. *Given a rooted C-heap $G = \langle N, E, L, R\rangle$, there is a unique rooted C-heap $G' = \langle N', E', L', R'\rangle$ such that:*

1. *R' is node n_0.*
2. *There is a bijective* renumbering *function "ren" such that G and G' are isomorphic up-to renumbering (i.e., $n_1 \to n_2 \in E$ iff $ren(n_1) \to ren(n_2) \in E'$).*
3. *$L' = \{fl : l \in L\} \cup \{bl : l \in L\}$.*
4. *Nodes in G' are listed in breadth-first order.*
5. *If an edge $e = n_1 \to n_2$ is labeled l in G, then:*
   (a) *if $ren(n_1) < ren(n_2)$, then $ren(n_1) \to ren(n_2) \in E'$ is labeled fl.*
   (b) *if $ren(n_1) \geq ren(n_2)$, then $ren(n_1) \to ren(n_2) \in E'$ is labeled bl.*

Proof. We begin by defining function *ren*. We define $ren(R) = N_0$. We then list the nodes of G in breadth-first order starting from R, and define $ren(n) = N_j$ if n is the j-th node in the breadth-first traversal of G. Notice that even though a node may have multiple parents, it has a unique $(f_1 + \cdots + f_k)$-min-parent; therefore, the traversal ordering is well-defined. Since each $n \in N$ is visited exactly once, *ren* is a bijection. We define $L' = \{fl : l \in L\} \cup \{bl : l \in L\}$. Edge labels are assigned as follows. If an edge $e = n_1 \to n_2$ is labeled l in G, then:

1. if $ren(n_1) < ren(n_2)$, then $ren(n_1) \to ren(n_2) \in E'$ is labeled fl.
2. if $ren(n_1) \geq ren(n_2)$, then $ren(n_1) \to ren(n_2) \in E'$ is labeled bl.

For each node $n \in N'$ and non-recursive field f, we define $f(n) = f(ren^{-1}(n))$. Uniqueness is guaranteed by construction.

Let M be an Alloy specification, and let M' be its instrumented counterpart. The following theorem shows that the instrumented specification has models that indeed satisfy the conditions of Thm. 1.

Theorem 2. *Let $Gr = \langle N, E, L, R\rangle$ be a model for M'. Then,*

1. *$R = N_0$,*
2. *Nodes in Gr are listed in breadth-first order from node N_0,*
3. *Given an edge $e = n_1 \to n_2 \in E$,*
   (a) *if $n_1 < n_2$, then e is labeled with a forward label.*
   (b) *if $n_1 \geq n_2$, then e is labeled with a backwards label.*

Proof. Fact "rootIsTheFirstNode" guarantees condition 1. We will now show that nodes are indeed listed in breadth-first order. The proof is by induction on the position in the breadth-first traversal of Gr. In position 0 we have node N_0, as required. Let us assume that for all positions $0 \leq j \leq k$, N_j is in position j. Let us consider n, the node in position $k+1$. Since nodes $N_0, \ldots, N_k$ are already listed, $n \geq N_{k+1}$. Let us suppose that $n > N_{k+1}$. Since n is reachable from the root, by fact "prefixComplete" N_{k+1} is also reachable from the root. Notice

that by fact "parentSmallerThanChildren", p_n, the $(f_1 + \cdots + f_i)$-min-parent of n, must be between N_0 and N_k. Let $p_{N_{k+1}}$ be the $(f_1 + \cdots + f_i)$-min-parent of N_{k+1}. If $p_n \neq p_{N_{k+1}}$, by fact "howToOrderTwoNodesWithDifferentParents", since n is listed before N_{k+1}, must be $p_n < p_{N_{k+1}}$. But then, $n < N_{k+1}$. This is a contradiction. In case n and N_{k+1} share the $(f_1 + \cdots + f_i)$-min-parent, since n is listed before N_{k+1}, must be $n < N_{k+1}$, a contradiction. Therefore, condition 2 is established.

Condition 3 is directly implied by fact "initialBounds".

The following theorem allows us to show that the instrumentation indeed reduces the state space.

Theorem 3. *Let $G = \langle N, E, L, R\rangle$ be a rooted C-heap. Let G' be constructed according to Thm. 1. Then, G is a model for the Alloy specification M if and only if G' is a model for the specification M'.*

Proof. Let *ren* be the renumbering function on signature C from Thm. 1. We extend *ren* as the identity over the remaining signatures. The proof proceeds by showing that the extension *ren*$'$ is an isomorphism between G and G'.

4 Using Symmetry Breaking to Compute Tight Bounds

In this section we present an important application of our symmetry breaking predicate. Let us consider our source Alloy model for binary trees. For some SAT-solvers (in particular for MiniSat [1]) the Alloy Analyzer allows, given an Alloy specification, to iterate through all the models for the specification. The Alloy Analyzer includes (when translating a specification to a propositional formula) its own symmetry breaking predicate. In Table 1 we compare the number of models generated from the source model, and from the instrumented model. The comparison is made varying over the number of nodes in the tree.

#TNode	3	4	5	6	7	8	9
Source	7	24	99	458	2320	12636	73713
Instrumented	5	14	42	133	431	1430	4862
% (I/S)	0.71	0.58	0.42	0.29	0.18	0.11	0.06

Table 1. Comparison of the number of generated models.

Notice that the percentage of instrumented models goes to 0 when the number of nodes goes to infinity. It is interesting to notice that despite the symmetry breaking predicate included in the Alloy Analyzer, isomorphic models are generated. For instance, for 3 nodes, the (isomorphic) models in Fig. 5 were generated (missing edges go to *null*).

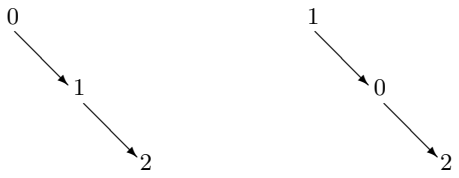

Fig. 5. Two isomorphic binary trees

If we look at the models generated from the instrumented Tree model, no tree can have an edge $TNode_1 \rightarrow TNode_0$. This would contradict fact "initialBounds". More important, there are other edges that cannot appear due to the ordering of nodes. Is it possible to have an edge $TNode_0 \rightarrow TNode_2$ labeled fleft?. The answer is "no". The ordering forces $TNode_0$ to be related either to *null* or to $TNode_1$.

While in the original Alloy model functions *left* and *right* are each one encoded using $n \times (n+1)$ propositional variables, due to the ordering of nodes we can remove arcs from relations. In order to determine wether an arc $TNode_i \rightarrow TNode_j$ can be part of field f, we perform the following analysis:

```
pred NiToNjInF[ ] {
   Ni+Nj in Tree.root.*(fleft + fright) and Ni->Nj in F
}
run NiToNjInF for exactly 1 Tree, exactly n TNode, exactly n Data
```

If the "run" command produces no instance, then the edge is unfeasible. Fact "initialBounds" makes the Alloy bounds to be recognized by KodKod. Therefore, as soon as an edge is determined to be unfeasible, it can be removed in fact "initialBounds". This reduced bounds, when received by KodKod, produce a SAT problem that involves a lesser number of propositional variables. In Section 5 we will discuss how to use ParAlloy in order to compute tight bounds in parallel.

5 Computing Tight Bounds with ParAlloy

Notice that computing the bounds requires checking, for each potential edge, its feasibility. For our Tree model, we must perform (for scope n for signature $TNode$), $n \times (n+1)$ analyses. Since all these analyses are independent, a naive algorithm consists on 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, while others may take hours. We now present a brief introduction to ParAlloy and describe the technique we currently use for computing bounds. In Section 5.1 we discuss optimizations to the technique.

ParAlloy is a tool that receives as input an Alloy model to be analyzed. Recall that Alloy translates a relational model to a propositional formula, which is

in turn fed to a SAT-solver. ParAlloy translates a relational model to a Boolean Expression Diagram (BED). BEDs are related to ROBDDs. But, unlike ROBDDs, parts of the structure may be kept as propositional formulas. Fig. 6 shows the structure of a BED in which some variables are as variables in an ROBDD (dashed arrows denote the *false* alternative, while full arrows signal the *true* alternative). BEDs provide operations for lifting a variable from the formula part, to the BDD part. In doing so, formulas are automatically split. If n variables are lifted, at most 2^n formulas hang from the BED (although experiments show that this number decreases significantly if appropriate variables are chosen to be lifted). This provides a controlled way of parallelizing the analysis of a single Alloy model. Given a certain infrastructure, just enough variables are lifted to guarantee that all the infrastructure will be used for SAT-solving each of the hanging formulas. If some of the analyses take longer than a threshold previously set, new variables are automatically lifted and simpler SAT problems are generated.

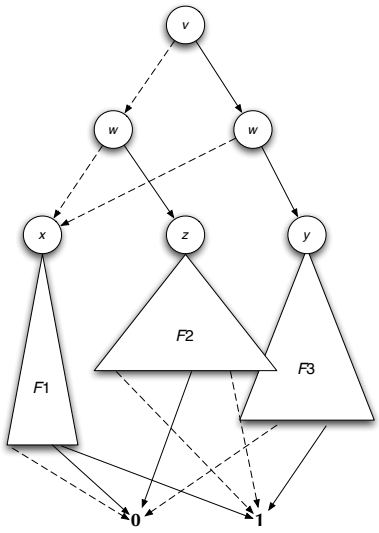

Fig. 6. The shape of a BED after some variables v, w, x, y and z were lifted. F_1, F_2 and F_3 are the hanging formulas.

For the sake of computing relational bounds, we modified ParAlloy as follows. ParAlloy receives now several Alloy models to be analyzed, one for each edge whose feasibility must be checked. Besides the threshold T_1 that is used by ParAlloy in order to determine when new variables must be lifted, a new threshold T_2 is added. The new threshold (which satisfies $T_2 < T_1$) is used as a timeout for the analysis. All the models are analyzed in parallel according to ParAlloy

policies. Those checks that exceed T_2 are stopped and left for a later stage. Each analysis that finishes as unsatisfiable tells us that an edge may be removed from the bound. Satisfiable checks tell that the edge cannot be removed. After all the models have been analyzed, we are left with a partition of the models in three sets: unsatisfiable checks, satisfied checks, and stopped checks for which we do not have a conclusive answer. We then refine the bounds (using the information from the unsatisfiable models) for the models whose checks were stopped. The formerly stopped models are sent again for analysis. This leads to an iterative process that after a few iterations converges to a set (possibly empty) of models that cannot be checked (even including the previous information) within the threshold T_2. For these models we use the full power of ParAlloy, including lifting variables repeatedly and splitting the models into simpler ones. In Section 6 we will present experiments showing the usefulness of this approach.

5.1 Optimized Algorithms for Bound Computation

In this section we present optimizations to the analysis process that have not yet been implemented. Since these optimizations present obvious advantages, they will be implemented in the near future. Notice first that the burden of the analysis procedure lies on the number of edges that must be checked. So far we are checking each edge independently. But, whenever a check for an edge $TNode_i \rightarrow TNode_j$ is satisfied, most probably more edges appear in the configuration returned by the check. These edges need not be analyzed, since they can obviously be satisfied by the model just found. Another optimization consists on, whenever the iterative process finishes and we are left with a nonempty set of edges that could not be analyzed, generate two models from each pending model. In one model we assume the edge cannot occur, and in the other we assume that the edge indeed occurs. Notice that in the propositional formula obtained from the models, the variable that models the edge is replaced by a constant value. Clearly the number of models grows exponentially, but we hope (and this is confirmed by the current experiments) that the number of checks that are pending at this stage will be small.

6 Experimental Results

In this section we present two nontrivial case-studies and present information related to their analysis. We will compute the bounds for a simpler model for doubly linked lists, and afterwards we will present a more complicated case-study based on AVL trees.

6.1 Computing Tight Bounds for Doubly Linked Lists

The (instrumented) invariant for Doubly Linked Lists is presented in Fig. 7. The experimental results are computed relative to a scope of 15 LNodes.

```
fact DLLInv {
   let next = fnext+bnext |
   let prev = fprev+bprev |
   // no cycles
   all n: thiz.head.*next-null | n !in n.next.*next and
   // symmetry
   all n: thiz.head.*next-null |
      (n.next=null implies thiz.tail=n) and
      (n.next!=null implies n.next.prev=n) and
   all n: thiz.tail.*prev-null |
      (n.prev=null implies thiz.head=n) and
      (n.prev!=null implies n.prev.next=n)
}
```

Fig. 7. Invariant for Doubly Liked Lists

In order to determine which edges remain, there is to perform 480 checks. For this case study, only 58 edges are considered feasible (12% of the original edges). The feasible edges of relation *fnext* are of the form

$$DLLNode_i \rightarrow (DLLNode_{i+1} + null) \quad \text{for } 0 \leq i \leq 14.$$

For relation *fprev* the feasible edges are

$$DLLNode_i \rightarrow (DLLNode_{i-1} + null) \quad \text{for } 0 \leq i \leq 14.$$

The backwards fields do not have admissible edges.

In Table **??** we present analysis times. Each line in the table refers to one phase in the iterative process. For each phase we report the time consumed for finishing that phase, the number of analyzed edges in that phase, and the number of timed-out edges that move to the next phase. The threshold time in this experiment was set to 40 seconds. The experiments were run on a cluster with Double Intel Dual Core Xeon processors running at 2.67 GHz. Each node has 2 Gb of RAM. Nodes are connected through a low latency Infiniband switch. For this experiment there were 48 available cores.

phase #	# edges	# solved	# timeouts	time
1	450	199	251	320”
2	251	155	96	203”
3	96	96	0	13”
Total	797	450	347	8’ 56”

Table 2. Analysis times for doubly linked lists

6.2 Computing Tight Bounds for AVL Trees

The (instrumented) invariant for AVL trees is presented in Fig. 8. The experimental results are computed relative to a scope of 15 AVLNodes.

```
fact AVLInv {
   let left = fleft + bleft |
   let right = fright + bright |
      all x: thiz.root.*(left+right) - null |
         // no cycles
         x !in x.^(left + right) and
         // ordered
         (all y: x.left.*(left+right) - null | lt[y.key ,x.key]) and
         (all y: x.right.*(left+right) - null | lt[x.key, y.key]) and
         // definition h
         (x.left=null and x.right=null) implies x.h=0 and
         (x.left=null and x.right != null) implies (x.h=1 and x.right.h=0) and
         (x.left != null and x.right=null) implies (x.h=1 and x.left.h=0) and
         (x.left != null and x.right != null) implies
            (x.h = add[larger[x.left.h, x.right.h],1] and
            (lte[-1,sub[x.left.h,x.right.h]] and lte[sub[x.left.h,x.right.h],1]))
}
```

Fig. 8. Invariant for AVL trees

In order to determine which edges remain, there is to perform 480 checks. For this case sutdy, only 129 edges are considered feasible (27% of the original edges). The backwards fields do not have any admissible edges.

In Table 3 we present analysis times. Each line in the table refers to one phase in the iterative process. For each phase we report the time consumed for finishing that phase, the number of analyzed edges in that phase, and the number of timed-out edges that move to the next phase. The threshold time in this experiment was set to 40 seconds. The experiments were run on a cluster with Double Intel Dual Core Xeon processors running at 2.67 GHz. Each node has 2 Gb of RAM. Nodes are connected through a low latency Infiniband switch. For this experiment there were 48 available cores.

7 Related Work

There are two approaches to reduction of symmetries that are close to ours. The first approach is the one used in Alloy 3, and discussed in [7]. Reduction of symmetries takes place at the relation level; that is, symmetries are reduced for each relation in isolation, and symmetries are broken only for relations having certain predefined properties (acyclic, permutation, functional, etc...). These properties

phase #	# edges	# solved	# timeouts	time
1	450	200	250	300”
2	250	72	178	210”
3	178	51	127	160”
4	127	35	92	165”
5	92	36	56	107”
6	56	24	32	61”
7	32	22	10	48”
8	10	8	2	45”
9	2	2	0	18”
Total	1197	450	747	18’ 34”

Table 3. Analysis times for AVL trees

have to be declared at modeling time for the translator to recognize them. In building the symmetry breaking predicate it does not use the constraints derived from the property under analysis. A similar situation occurs with the symmetry breaking predicate included in KodKod [8]. Notice that this is the expected scenario when designing symmetry breaking predicates for arbitrary models. Our models have restrictions and this allows us to use the particular symmetry breaking predicates presented in this article. Moreover, the methodology we devised for using the symmetry breaking predicate allows us to use the properties of the model under analysis. This allows us to prune the upper bounds of relations significantly better than the previous predicates. This is shown for instance in Table 1. An even closer approach to ours is presented in [5], where the idea of using predicates that linearize the data structure is presented. The proposal does not present a generic predicate to this end, but rather recommends to look for new predicates depending on the structure. It does not generate bounds as we do.

8 Conclusions and Further Work

SAT-based automated program verification has clear limitations. Reasonable pieces of code yield very hard SAT problems. It is therefore essential to find optimizations that will improve the analysis. We present one such optimization. Finding tight relational bounds leads to a SAT problem that involves less variables. For the case studies we presented, the number of removed propositional variables is very significative. This has a huge impact in SAT-solving.

In Section 5.1 we propose some optimizations that will be implemented before the workshop takes place. Also, at the workshop we will report on the impact this technique has on automated verification of object-oriented programs.

References

1. N. Een and N. Sorensson. *An extensible SAT solver*. In International Conference on Theory and Applications of Satisfiability Testing, pages 502–518, May 2003.
2. M. Frias, J.P. Galeotti, C. López Pombo and N. Aguirre, *Efficient Analysis of DynAlloy Specifications*, to appear in ACM Transactions on Software Engineering and Methodology (TOSEM), ACM Press.
3. Galeotti J.P., and Frias M.F., *DynAlloy as a Formal Method for the Analysis of Java Programs*, in Proceedings of IFIP Working Conference on Software Engineering Techniques (SET06), Warsaw, 2006, Springer.
4. Jackson, D. *Software Abstractions.* 2006, The MIT Press.
5. Khurshid S., Marinov D., Shlyakter I., Jackson D., *A Case for Efficient Solution Enumeration*, in Proceedings of the 6th. International Conference on Theory and Applications of Satisfiability Testing (SAT 2003).
6. Leavens, G., Baker, A., and Ruby, C. *JML: a Notation for Detailed Design.* In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), Behavioral Specifications for Businesses and Systems, chapter 12, 1999, pages 175–188.
7. Shlyakhter I., *Generating Effective Symmetry Breaking Predicates for Search Problems*, in Discrete Applied Mathematics, VOl. 155, No. 12, pp. 1539–1548, 2007.
8. E. Torlak and D. Jackson. *Kodkod: A Relational Model Finder. Tools and Algorithms for Construction and Analysis of Systems* (TACAS '07). Braga, Portugal, March 2007.