BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support

Nicolás Rosner, Jaco Geldenhuys, Nazareno M. Aguirre, Willem Visser, and Marcelo F. Frias

Abstract—Lazy Initialization (LI) allows symbolic execution to effectively deal with heap-allocated data structures, thanks to a significant reduction in spurious and redundant symbolic structures. Bounded lazy initialization (BLI) improves on LI by taking advantage of precomputed relational bounds on the interpretation of class fields in order to reduce the number of spurious structures even further. In this paper we present bounded lazy initialization with SAT support (BLISS), a novel technique that refines the search for valid structures during the symbolic execution process. BLISS builds upon BLI, extending it with field bound refinement and satisfiability checks. Field bounds are refined while a symbolic structure is concretized, avoiding cases that, due to the concrete part of the heap and the field bounds, can be deemed redundant. Satisfiability checks on refined symbolic heaps allow us to prune these heaps as soon as they are identified as infeasible, i.e., as soon as it can be confirmed that they cannot be extended to any valid concrete heap. Compared to LI and BLI, BLISS reduces the time required by LI by up to four orders of magnitude for the most complex data structures. Moreover, the number of partially symbolic structures obtained by exploring program paths is reduced by BLISS by over 50 percent, with reductions of over 90 percent in some cases (compared to LI). BLISS uses less memory than LI and BLI, which enables the exploration of states unreachable by previous techniques.

Index Terms—Symbolic execution, lazy initialization, tight field bounds, Symbolic PathFinder

1 Introduction

Determining to what extent a software artifact is correct is among the most challenging problems in software engineering. Traditional testing is a widely adopted approach to guaranteeing software correctness, but its well-known limitations threaten its effectiveness as a bug-finding technique. Therefore, more thorough program analysis techniques, which may offer greater levels of confidence (often enhancing or complementing traditional testing) constitute an important research topic.

One technique that offers better guarantees of correctness is model checking [5]. Java PathFinder (JPF) [22] is a well-known tool, based on this technique, that targets Java source code and is capable of finding bugs in both sequential and multithreaded programs. Moreover, through an extension called Symbolic PathFinder (SPF) [17], [23], the tool is able to automatically generate test cases, search for violations of user-provided assertions or uncaught exceptions, handle arithmetic constraints, complex data structures and rich constraints on the program inputs.

- *N. Rosner is with the Department of Computer Science, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina. E-mail: nrosner@dc.uba.ar.*
- *J. Geldenhuys and W. Visser are with the Department of Computer Science, University of Stellenbosch, Stellenbosch, South Africa. E-mail: jaco@cs.sun.ac.za, willem@gmail.com.*
- *N. M. Aguirre is with the Department of Computer Science, FCEFQyN, Universidad Nacional de Rio Cuarto, and CONICET, Río Cuarto, Argentina. E-mail: naguirre@dc.exa.unrc.edu.ar.*
- *M.F. Frias is with the Department of Software Engineering, Instituto Tecnológico de Buenos Aires, and CONICET, Buenos Aires, Argentina. E-mail: mfrias@itba.edu.ar.*

Manuscript received 2 May 2014; revised 25 Nov. 2014; accepted 3 Dec. 2014. Date of publication 0 . 0000; date of current version 0 . 0000. Recommended for acceptance by A. Roychoudhury. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSE.2015.2389225

SPF combines *symbolic execution* [15] with model checking and constraint solving. Symbolic execution, a well-established program analysis technique, traverses the different paths in a program using symbolic inputs. Unlike the concrete states in JPF, states in SPF are *symbolic*. Symbolic approaches to systematically exploring program paths have proved effective for verification, as well as for automated test input generation by solving the path constraints obtained during the exploration. When using these symbolic approaches [2], [3], [6], [17], verifying code that manipulates dynamically allocated data structures is significantly more difficult than verifying code dealing with basic data types (the traditional target of symbolic execution). To effectively handle heap-allocated data structures, SPF generalizes symbolic execution by introducing *Lazy Initialization* (LI) [14]: it constructs the heap as the program paths are explored, and defers concretization of symbolic heap object attributes as much as possible.

LI has two important properties. Firstly, it produces symbolic heaps that are pairwise non-isomorphic. The number of heaps over which a method must be symbolically executed is greatly reduced by the elimination of symmetric structures, while guaranteeing that no relevant states are missed. Secondly, LI exploits any method precondition provided, by filtering out any heaps that violate it.

To improve symbolic execution, LI can be enhanced with the use of precomputed, relational *field bounds* [11]. Intuitively, field bounds restrict the number of choices that LI needs to consider when it is forced to concretize a part of the heap. In [12] we realized this idea using translation of annotated code (TACO) bounds [10] (which are discussed in Section 2.2), introducing bounded lazy initialization (BLI) and obtaining significant speedups with respect to LI.

In this paper we present a set of novel techniques that build on LI and BLI. They incorporate *bound refinement*

[refined bounded lazy initialization (RBLI)] and *satisfiability checks* [bounded lazy initialization with SAT support (BLISS, BLISSDB)]. In the former, field bounds are refined by leveraging information from already-concretized fields; this makes it possible to further prune the alternatives to be considered during symbolic execution. In the latter, auxiliary satisfiability checks are employed to determine the feasibility of partially symbolic structures, forcing the symbolic execution process to backtrack as soon as a partially symbolic candidate is found impossible to extend.

BLISS, as these techniques are collectively called, allows us to obtain speedups of more than 100X over LI in many cases. For example, the analyses of the contains methods from classes TreeSet and AvlTree achieved speedups of 14X and 188X over LI, respectively. Furthermore, our techniques also provide advantages in the context of automated test input generation, a task for which, as we mentioned, symbolic execution is particularly effective [3]. Indeed, BLISS can usually reduce the number of partially symbolic structures collected by over 50 percent, with reductions of over 90 percent in some cases, compared to LI. Since these partially symbolic structures must be fully concretized (using SMT solving) to produce actual test inputs, and since BLISS only removes spurious cases, the technique impacts test input generation time while retaining the same coverage obtained by LI. For instance, for the above-mentioned contains methods, the sets of partially symbolic structures obtained by BLISS are 12.5 and 1.6 percent as large as the ones obtained using LI, respectively.

The main contributions of this paper are:

1) We introduce refined bounded lazy initialization, a sound and complete optimization of BLI. RBLI requires the existence of relational field bounds (as introduced to SPF in BLI) and is often responsible for most of the speedup observed.
2) We introduce bounded lazy initialization with SAT support, an additional optimization which builds upon the existence of refined bounds as produced by RBLI, and makes use of user-provided class representation invariants. We show that BLISS is sound and complete, and that it is often responsible for most of the reduction in the number of partially symbolic structures obtained during systematic path exploration.
3) We optimize BLISS by caching SAT results. BLISS typically produces a large number of short computations, and a significant portion of these can be reused in later, related analyses. Hence, we cache SAT results in a Redis [18] database, leading to an optimized version that we call BLISSDB.
4) We evaluate our techniques on a benchmark consisting of 32 methods from six well-known collection classes, and show that the combination of the techniques can yield significant speedups as well as considerable test suite size reduction.

This paper is organized as follows. Section 2 describes SPF and briefly reviews our previous work on LI and BLI, the existing techniques for symbolically executing code that handles heap-allocated data structures. In Section 3 we present RBLI and prove its soundness and completeness. In Section 4 we introduce BLISS, prove its soundness and completeness, and present BLISSDB. Section 5 contains an evaluation of RBLI, BLISS and BLISSDB on several implementations of collection classes, some of which have been previously used to evaluate SPF. Lastly, Section 6 discusses related work, and in Section 7 we present our conclusions and proposals for further work.

2 Symbolic PathFinder and (Bounded) Lazy Initialization

Java PathFinder is a flexible tool for software analysis. Its core is a virtual machine (VM) for Java byte code. Unlike a standard Java VM, the JPF VM is capable of backtracking. The tool identifies program statements that lead to alternative branches, and systematically explores those alternatives. Besides branching that arises as a consequence of thread scheduling, the main source of nondeterminism is program inputs. When these inputs involve—for instance—integer variables, this may lead to path explosion (often called state space explosion).

Symbolic execution [15] collapses families of executions by replacing concrete values with symbolic ones. Whenever branching conditions are encountered in the program, constraints are collected to reflect the decisions that were taken; the conjunction of constraints along one program path is referred to as the *path condition* for that path. Such conditions are checked for feasibility using constraint solvers (typically SMT solvers), and when one is found to be infeasible, the underlying model checker driving the symbolic execution backtracks and explores other paths. This systematic exploration of paths can be used for verification and bug finding. Moreover, the path conditions obtained in the exploration of program paths can be solved to find concrete inputs that will drive an execution down the corresponding paths, thus leading to a mechanism for automated white-box test input generation.

In this paper we present improvements over existing work for the symbolic analysis of code handling dynamically allocated data structures. Therefore, in Section 2.1 we summarize Lazy Initialization [14], the technique currently used by SPF for exploring such data structures. In [12] we introduced a first improvement over Lazy Initialization; in order to discuss this technique (which is called bounded lazy initialization) in Section 2.3, we first introduce in the concept of TACO bounds in Section 2.2.

2.1 Lazy Initialization

For heap-allocated structures, SPF uses *Lazy Initialization* [14]. LI keeps structures partially symbolic; if an object's attribute f is still symbolic, it will be made concrete whenever the execution of the program under analysis attempts to access its value. This on-demand concretization explains the "lazy" appellative of the algorithm. The concretization process considers three possibilities: f is initialized with null, f is initialized with a previously introduced concrete object, or f holds a reference to a newly introduced concrete object whose attributes are all symbolic. These choices are systematically explored by the underlying model checker. A pseudocode description of LI is shown in Algorithm 1, originally presented in [14]. Notice that the

```
public class BinTree {
    Node root;
}

public class Node {
    Node left;
    Node right;
}
```

Fig. 1. An implementation of heap-allocated binary trees.

code under analysis need not execute on a purely symbolic structure; it may execute on an input that is partially symbolic and for which symbolic parts are explored using LI. This same property holds for the techniques we will introduce in further sections.

Algorithm 1. Pseudocode of the Lazy Initialization algorithm.

```
if (f in uninitialized) then
  if (f is reference field of type T) then
    nondeterministically initialize f to
    1. null
    2. a new object of class T (with uninitialized fields)
    3. an object created during a prior
                          initialization of a field of type T
    if (method precondition is violated) then
      backtrack();
    end
  end
  if (f is primitive field) then
    initialize f to a new symbolic value of appropriate type
  end
end
```

Fig. 2b shows some of the alternatives explored by the LI algorithm when executing the traverse algorithm from Fig. 2a on a binary tree, with binary trees defined by classes BinTree and Node as shown in Fig. 1. Executing traverse on structure 1 from Fig. 2b reaches the branching condition "`right != null`". Therefore, field right must be concretized. This leads to the generation of structures 2-4 in Fig. 2b. Let us continue with structure 4. Upon execution of the statement "`right.traverse();`" on this structure, the recursive invocation of traverse leads us, once again, to the concretization of N1.right. This time, 4 alternatives are generated. Notice that amongst these structures, some are clearly invalid due to the presence of loops (this is the case for instance for structures 3, 6 and 7). In order to prune such invalid structures at an early stage, LI resorts to *preconditions*. We discuss below what preconditions are, and how they are applied during LI.

2.1.1 Preconditions and Lazy Initialization

A *precondition* for a method m is a condition that is assumed to be true before the execution of m. Such conditions are used by the class designer/programmer to characterize those input states in which the method is expected to behave as intended. For example, the method for traversing a binary tree depicted in Fig. 2a requires the input structure to be a binary tree, and in particular, to be non-null and acyclic, since the algorithm might otherwise perform a null dereference or get stuck in an infinite loop. In object-oriented programming, one part of a method's

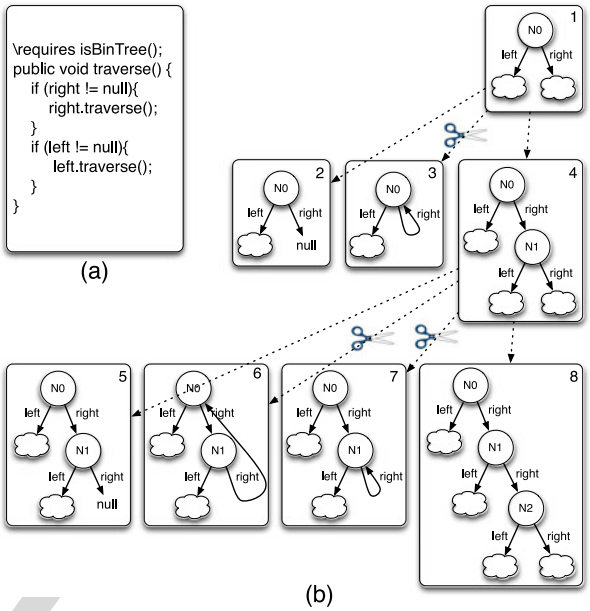


Fig. 2. Method traverse on binary trees, and some of the structures generated by LI along its symbolic execution.

precondition is usually the *representation invariant* of the method's parameters (including that of the implicit `this` object on which the method is executed). A representation invariant, also called class invariant, is a condition that accompanies a class, which must be established by its constructors and preserved by its public methods. Thus, such invariants characterize properties of valid instances of the class. For instance, for heap-allocated binary trees, the representation invariant would specify that the structure is indeed a tree (acyclic, with every node having exactly one parent except for the root).

Besides being useful as program documentation, representation invariants (and, more generally, preconditions) can be reflected programmatically as imperative routines that check whether the invariant or precondition holds. Imperative representation invariants are usually referred to as repOK routines.

As explained in [14], and as illustrated in Algorithm 1, LI requires an imperative precondition for the program or method under test. But not just any precondition will work properly with LI: the average repOK routine, for instance, is not necessarily prepared to deal with the fact that it might run into parts of the structure that are still symbolic. In other words, LI assumes the existence of an imperative precondition that has been properly adapted to deal with partially symbolic structures. We shall refer to such preconditions as *hybrid*, since they can be applied to structures involving both concrete and symbolic values.

A hybrid precondition could be a straightforward adaptation of the original concrete precondition: one that attempts to detect ill-formed structures, but returns true as soon as it runs into a symbolic value that it does not know how to handle. Of course, it could also be a much more sophisticated routine, carefully designed by the user with symbolic execution in mind—for instance, one that backtracks whenever symbolic values are found, and tries to detect ill-formedness later on. This raises an important trade-off. The former approach is a simple

```
public boolean acyclicConcrete() {
   Set<BinTreeNode> visited = new HashSet<BinTreeNode>();
   List<BinTreeNode> pending = new ArrayList<BinTreeNode>();
   BinTreeNode root = this.root;
   visited.add(root);
   pending.add(root);
   while (!pending.isEmpty()) {
      BinTreeNode node = pending.remove(0);
      BinTreeNode left = node.left;
      if (left != null) {
         if (!visited.add(left)) {
            return false;
         }
         pending.add(left);
      }
      BinTreeNode right = node.right;
      if (right != null) {
         if (!visited.add(right)) {
            return false;
         }
         pending.add(right);
      }
   }
   return true;
}

public boolean acyclicHybrid() {
   if (this == SYMBOLIC_BinTree)
      return true;

   Set<BinTreeNode> visited = new HashSet<BinTreeNode>();
   List<BinTreeNode> pending = new ArrayList<BinTreeNode>();
   BinTreeNode root = this.root;
   if (root == SYMBOLIC_BinTreeNode)
      return true;

   visited.add(root);
   pending.add(root);
   while (!pending.isEmpty()) {
      BinTreeNode node = pending.remove(0);
      BinTreeNode left = node.left;
      if (left != null && left != SYMBOLIC_BinTreeNode) {
         if (!visited.add(left)){
            return false;
         }
         pending.add(left);
      }
      BinTree right = node.right;
      if (right != null && right != SYMBOLIC_BinTreeNode) {
         if (!visited.add(right)){
            return false;
         }
         pending.add(right);
      }
   }
   return true;
}
```

Fig. 3. A concrete precondition and its hybrid counterpart.

over-approximation that may return false positives, which could slow down the analysis, but it has the enormous advantage of being fully automatic. The latter approach, on the other hand, is more specific, less prone to false positives and thus potentially more scalable, but it requires expert human intervention and bears considerable risk of introducing new errors. Ensuring that a handcrafted hybrid precondition is correct with respect to the original concrete one would become a nontrivial problem on its own right. This is why we chose the former (i.e., to systematically derive a conservative hybrid precondition from an available concrete one) as our default course of action, and for experimental evaluation. For example, Fig. 3 shows an acyclicConcrete method that checks whether a fully concrete structure is acyclic, and a hybrid version thereof, acyclicHybrid, which will admit partially symbolic structures. The hybrid version includes special constants and associated boilerplate code in order to handle symbolic values from each of the types involved.

There are some limitations to what can be pruned using hybrid preconditions. These limitations stem from the fact that LI only sets the values of accessed fields of reference types. Primitive-typed fields, however, obtain their values from the solutions to path conditions, which are computed by constraint solvers. Unfortunately, these primitive values usually cannot be used within hybrid preconditions. To illustrate this fact, let us enrich our BinTree class with an `int` field named key, and consider again method traverse from Fig. 2a, enriching its precondition with a new constraint that requires the root's key to have value 0. Recall that hybrid preconditions are used to prune symbolic execution, by forcing the process to backtrack when a symbolic instance is found not to be obtainable from an initial heap satisfying the hybrid precondition. Since traverse does not access field key, the constraint solver may assign arbitrary values to root.key. In particular, if it were to assign a nonzero value to said field, this would lead to pruning a valid partially symbolic tree, as well as all of its concretizations, thus turning LI into an incomplete technique.

One might argue that, rather than using the constraints on primitive values to prune symbolic execution, such constraints could be used to enhance path conditions, thus narrowing the space of solutions found by constraint solvers. However, let us recall that preconditions are imperative in this context, and therefore cannot be directly conjoined with path conditions. One way of "conjoining" imperative preconditions with path conditions is by sequentially composing the imperative precondition with the program under analysis. Although this does achieve the desired goal of integrating the constraints on the primitive values into the path conditions, it severely hinders scalability: in the code resulting from such a composition, the imperative precondition "prefix" will typically force an enumeration of all valid concrete (or nearly concrete) instances prior to the execution of the routine under analysis, which defeats the purpose of symbolically executing said routine.

Yet another alternative would be to require a *declarative* precondition to be provided, so that it can be directly conjoined with path conditions. This approach has serious disadvantages as well. If the whole declarative precondition is included in each and all path conditions, their sizes are substantially increased, making them exceed the capabilities of the constraint solver sooner, thus diminishing scalability. If, instead, the path conditions are solved before symbolic execution, so that they need not be carried along all symbolic paths, then we end up, as in the aforementioned case, enumerating all valid concrete (or nearly concrete) instances.

2.2 TACO and Field Bounds

TACO [10], [11] is a tool for bounded program verification that targets Java code annotated with JML [9] contracts. In order to verify the correctness of a Java program, i.e., that it does not violate its contract and does not raise unhandled exceptions, it requires the engineer to provide a *scope*, which consists of a maximum number of iterations and object instances for the classes involved. It then checks the correctness of the program within the provided scope—that is, it checks that no execution involving at most as many objects and iterations as prescribed by the scope can violate the contract or raise an unhandled

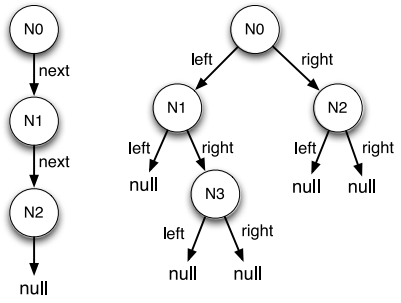


Fig. 4. Object labeling according to TACO's symmetry breaking.

- root $\subseteq$ {**N0**, null}
- left $\subseteq$ {(N0, null), (**N0**, **N1**), (**N1**, **null**), (N1, N2), (N1, N3), (**N2**, **null**), (N2, N3), (**N3**, **null**)}
- right $\subseteq$ {(N0, null), (N0, N1), (**N0**, **N2**), (N1, null), (N1, N2), (**N1**, **N3**), (**N2**, **null**), (N2, N3), (**N3**, **null**)}

(a)

- root $\subseteq$ {N0, null}
- left $\subseteq$ {(N0, null), (N0, N1), (N1, null), (N1, N3), (N2, null), (N3, null)}
- right $\subseteq$ {(N0, null), (N0, N2), (N1, null), (N1, N3), (N2, null), (N3, null)}

(b)

Fig. 5. Tight relational bounds automatically computed by TACO for binary trees (a) and for complete binary trees (b), using a scope of up to 4 nodes.

exception. To achieve this, TACO translates the program and its declarative specification into a propositional formula, which is solved using off-the-shelf SAT-solvers. Essentially, satisfying valuations of the resulting formula correspond to program executions violating the program specification; thus, unsatisfiability of the formula means that the program is correct within the given scope. Notice that these declarative specifications most times include a declarative representation invariant, which is part of the method precondition.

In TACO, the encoding of bounded program correctness as a satisfiability problem involves characterizing heap states as relations. Given a class C, a class field f of type C' defined in C can be represented in a given program state as a total function f mapping object references from C to C'. Notice that properties of the state, including the maximum number of objects of each class (i.e., the scope), may make some tuples of $C \times C'$ infeasible as part of field f. In particular, if the state is assumed to satisfy constraints (e.g., the states prior to the execution of the code under analysis are assumed to satisfy a precondition), all tuples corresponding to ill-formed structures will necessarily be absent from f in that state. Furthermore, if symmetry breaking is imposed by enforcing a canonical ordering on the way references are stored in the heap model (see [11] for a careful introduction), then structures that do not comply with this canonical ordering are dismissed, and the number of tuples allowed in the relations that bound the fields can be significantly reduced. TACO field bounds capture precisely these feasible cases. A field bound for a field f of type $C \rightarrow C'$ is a subset $U_\mathsf{f} \subseteq C \times C'$, such that every tuple t that is not U_f cannot correspond to the contents of f in any valid instance of C within scope k. By *valid instance* we mean an instance that satisfies the corresponding specification and symmetry-breaking constraints associated with the field. Essentially, tuples that are absent from the upper bound U_f are infeasible, i.e., are guaranteed not to belong to *any* valid instance. Note that $C \times C'$ is a field bound for a field f of type $C \rightarrow C'$, although it is not necessarily the *tightest* possible bound (i.e., the one containing the smallest possible subset of tuples). The tighter a bound, the better, since it provides more information about infeasible cases for the corresponding field. While a thorough description of our symmetry-breaking process is given in [10], we emphasize that the induced canonical ordering labels object identifiers in accordance with a breadth-first traversal of the memory heap. Fig. 4 shows how a singly-linked list and a binary tree are labeled. It is also worth mentioning that TACO field bounds, that is, those that were automatically computed using the approach put forward in [10], are the tightest possible bounds for the data structures that were studied in [10] (which include those analyzed in this paper): they exclude every tuple that can be proved infeasible for the corresponding class invariants and scopes.

In order to illustrate TACO field bounds, consider again binary trees, as defined in Fig 1. The representation invariant for this structure, which is expected to be part of the precondition of any method handling binary trees, requires the heap structure starting at the root to be a tree. Symmetry breaking forces tree node reference labels to be assigned in breadth-first order. Suppose that the scope is 4 (i.e., we only consider trees containing at most 4 nodes). The tightest possible field bounds for fields root, left and right are shown in Fig. 5a, and are exactly those computed by TACO. Notice that the binary tree illustrated in Fig. 4 satisfies the constraints (valid tree, with nodes labeled in breadth-first order, and within scope 4); the tuples in the bounds that are involved in this tree are highlighted in Fig. 5a.

Clearly, for a given scope, the tightest field bounds are determined both by symmetry breaking and by the class invariant. Therefore, if we considered a stronger class invariant, the corresponding field bounds would be more restrictive (that is, they would contain fewer tuples). Continuing with our example, if we used a stronger class invariant, for instance, one characterizing *complete* binary trees (complete up to the penultimate level, and such that nodes on the last level are located as far to the left as possible), then the corresponding bounds would be the ones given in Fig. 5b.

2.3 Bounded Lazy Initialization

Bounded lazy initialization [12] is an optimization of LI that leverages the availability of TACO bounds. Essentially, TACO bounds are used to reduce the number of alternatives that need to be explored during symbolic execution, avoiding the generation of some of the structures that LI would produce. Instead of being labeled with object identifiers, nodes in partially symbolic structures are labeled with *sets* of object identifiers, in accordance with the bounds. Given a partially symbolic structure S and a node N in S whose label is a set l_N of identifiers, this set intuitively denotes the set of object identifiers that could potentially be assigned to node N in a concrete structure extending S. Naturally, in each concrete structure, a single identifier is assigned. But, since TACO picks identifiers in a canonical way that is consistent with a breadth-first traversal of the structure, a node may

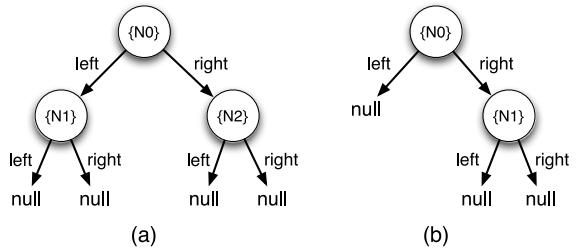


Fig. 6. Shifting of node identifiers due to null values in the BFS traversal.

receive different identifiers in different structures depending, for instance, on the number of null values found before N along the traversal. Fig. 6 shows a pair of concrete binary trees. Each node is assigned an identifier picked from the set $\{\mathsf{N0}, \mathsf{N1}, \mathsf{N2}\}$ under the constraint that the BFS traversal of the trees must produce a sorted sequence (we assume $\mathsf{N0} < \mathsf{N1} < \mathsf{N2}$). Note that, in Fig. 6a, the rightmost node has been assigned identifier N2; this is the only possibility that respects the BFS traversal. In Fig. 6b, the rightmost node is assigned identifier N1. This is due to the presence of null as the left subtree of the root node, which shifts the available node identifiers.

Root nodes receive as their label set their corresponding field bounds, without null. Field bounds are also involved in the definition of labels for non-root nodes. Given a node N with label l_{N}, we define its target label set through field f as the set $\mathrm{label}(\mathsf{N}, \mathsf{f})$ characterized by the following expression:

$$\mathrm{label}(\mathsf{N}, \mathsf{f}) = \left(\bigcup_{n \in l_{\mathsf{N}}} n.U_{\mathsf{f}} \right) - \mathsf{null} \ ,$$

where U_{f} is the field bound for f. Notice that null is never part of the label set of a node. This is because only *concrete* nodes are assigned label sets, and the label corresponds to the identifiers that this node may receive. When the value of attribute f for node N has to be concretized following the BLI algorithm, we consider the following alternatives:

- N.f is set to null,
- N.f points to an existing concrete node N′ if the latter has a label set that has a nonempty intersection with $\mathrm{label}(\mathsf{N}, \mathsf{f})$, or
- N.f points to a newly introduced concrete node N′ whose label is defined as $\mathrm{label}(\mathsf{N}, \mathsf{f})$ if the latter set is nonempty.

The cases in which pruning takes place are the second and the third. Let us describe this in more detail. As we said, the label set associated with a node captures the alternative identifiers that the node may adopt. When a previously introduced node N_t is being considered as the f field of another node N_s, this case only makes sense as an option if the label set of N_t has some intersection with the possible values reachable from N_s's label set through f, according to the bounds. Similarly, when a node's label set is empty, it means that no identifier can be assigned, which deems the extension infeasible. Algorithm 2 shows the pseudocode of the BLI algorithm. Notice how the alternatives for the initialization of fields with previously visited nodes is reduced in this algorithm, compared with LI

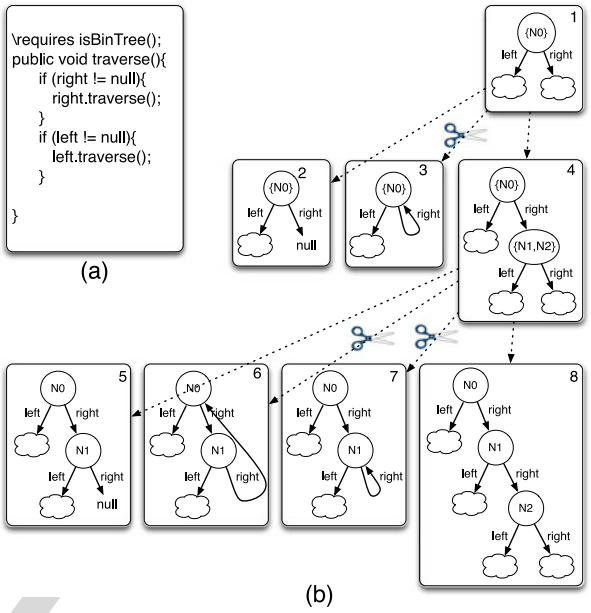


Fig. 7. Some of the structures generated by BLI during execution of method traverse.

Fig. 7 shows how the generation of structures from Fig. 2 is carried out in the context of BLI when the bound for binary trees (Fig. 5a) is considered. In the example we begin with the heap root labeled {N0} (the other possibility being for the root to be null). When accessing field right, BLI only generates structures 2 and 4. Structure 3 is never generated because the label for the root node (set {N0}) and the set $\mathrm{label}(\mathsf{N0}, \mathsf{right})$ (set $\{\mathsf{N1}, \mathsf{N2}\}$) do not intersect. A similar reasoning explains why structures 6 and 7 are not generated. Notice that in these cases we have only prevented the generation of structures that would be deemed redundant by LI as well upon execution of the precondition. We have only saved the time that would have been spent in the execution of the precondition.

Algorithm 2. Pseudocode for the bounded lazy initialization algorithm.

```
if (f is uninitialized) then
  if (f is reference field of type T) then
    nondeterministically initialize f to
    1. null
    2. a new object n of class T (with uninitialized fields)
                  and label(n) := label(this,f),
                  if label(this,f) is nonempty
    3. an object x created during a prior
                  initialization of a field of type T
                such that label(this).intersects(label(x))
    if (method precondition is violated) then
        backtrack();
    end
  end
  if (f is primitive field) then
    initialize f to a new symbolic value of appropriate type
  end
end
```

BLI may also prune subtrees that would not be pruned by LI. Let us consider structure 4 from Fig. 2. If we now

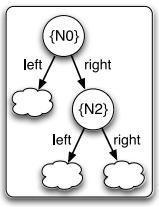


Fig. 8. A version of structure 4 extracted from Fig. 2, labeled using bounds for complete binary trees.

use bounds for complete binary trees (see Fig. 5b), an accordingly labeled version of structure 4 is shown in Fig. 8. Notice that LI would extend structure 4 to generate structure 8, and would even attempt further extensions. Instead, since $\text{label}(\mathsf{N2}, \mathsf{right}) = \emptyset$, the only possibility for concretizing N2.right according to BLI is to assign null. Therefore, a unique extension, namely, structure 5, will be produced by BLI.

BLI is sound and complete with respect to LI, provided that field bounds are correct (they only exclude infeasible tuples). BLI's soundness and completeness with respect to LI mean that a valid structure (one that satisfies the existing class invariant), is generated by BLI if and only if it is generated by LI [12].

Theorem 1. *Let M be a method under analysis. A valid structure S is generated along the symbolic execution of method M using LI if and only if it is generated along the symbolic execution of M using BLI.*

Proof. $\Rightarrow$) Let us assume S is not generated via BLI and let us arrive at a contradiction. As discussed above, there are two situations in which S may be discarded by BLI, namely,

- the concretization of N.f is a new node whose label set is empty, or
- the concretization of N.f is a previously introduced node whose label set does not intersect $\text{label}(\mathsf{N}, \mathsf{f})$.

Regarding the first case, since S is pruned by BLI, there is a symbolic execution step in which the concretization of attribute f from node N leads to a new node N′ using LI, but $\text{label}(\mathsf{N}, \mathsf{f}) = \emptyset$ and S is pruned using BLI. But, since bounds are correct and S is a valid structure, $(\mathsf{N}, \mathsf{N}') \in U_{\mathsf{f}}$ and therefore $\text{label}(\mathsf{N}, \mathsf{f}) \neq \emptyset$ (a contradiction).

Regarding the second case, let S_0 be a symbolic structure whose concretization using LI leads to S, and such that it is discarded by BLI when node N is made to point to a previously existing node N′ with $\text{label}(\mathsf{N}, \mathsf{f})$ disjoint from the label set of N′. Notice that for an arbitrary node $\mathsf{N_0}$ in S_0, its label set is the set of node identifiers that can be assigned to node $\mathsf{N_0}$ along BFS traversals of full concretizations of S_0. Let LS_0 and LS_1 be the label sets for nodes N and N′, respectively. Since S is a valid structure, let n_0, n_1 be the identifiers assigned to nodes N and N′ in the BFS traversal of S, respectively. Clearly, $n_0 \in LS_0$ and $n_1 \in LS_1$. Also, in S, $n_0.\mathsf{f} = n_1$. Thus, $\langle n_0, n_1 \rangle \in \mathsf{U_f}$. But then, $n_1 \in \text{label}(\mathsf{N}, \mathsf{f})$ and $\text{label}(\mathsf{N}, \mathsf{f}) \cap LS_1 \neq \emptyset$ (a contradiction).

$\Leftarrow$) Trivial (since BLI is more restrictive than LI). □

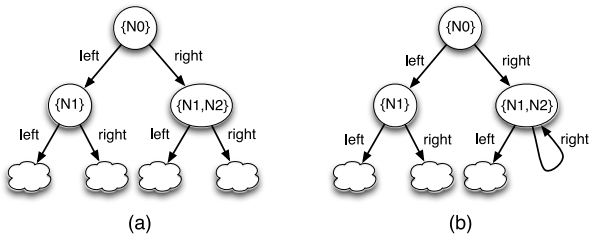


Fig. 9. A binary tree and an extension generated by BLI, yet pruned by RBLI.

3 Refined Bounded Lazy Initialization

Refined bounded lazy initialization is the first technique that we introduce in this paper, building upon BLI. We will show that the proposed technique is sound and complete, and will also set the basis for the experimental evaluation that will be reported in Section 5.

Notice that the number of structures generated by BLI during concretization is directly related to its label set. A label set containing fewer identifiers would usually produce fewer candidate structures. Let us consider the binary tree depicted in Fig. 9a. Let N denote the node whose label is the set $\{\mathsf{N1}, \mathsf{N2}\}$. According to the bounds for binary trees (see Fig. 5a), $\text{label}(\mathsf{N}, \mathsf{right}) = \{\mathsf{N2}, \mathsf{N3}\}$. Since $\{\mathsf{N1}, \mathsf{N2}\} \cap \{\mathsf{N2}, \mathsf{N3}\} \neq \emptyset$, the structure depicted in Fig. 9b is generated when attribute right is concretized following BLI (as well as when LI is used). In the remaining parts of this section we will argue that the generation of such candidates may be safely prevented.

A closer look at the reason why the structure depicted in Fig. 9b was generated shows that if identifier N1 were not part of the label for node N (i.e., if the label for node N was $\{\mathsf{N2}\}$), then $\text{label}(\mathsf{N}, \mathsf{right})$ would be $\{\mathsf{N3}\}$. Therefore, since $\{\mathsf{N2}\} \cap \{\mathsf{N3}\} = \emptyset$, the structure would not be generated. Recalling the explanation from Section 2.3 for having sets of identifiers (rather than just identifiers) as labels, we note that in a binary tree that respects the symmetry breaking imposed by TACO, node N0.right may be assigned identifiers N1 or N2. But, since identifier N1 has been already assigned to a different node prior to N in the breadth-first search traversal of the structure, it can be removed in this partially symbolic structure from set $\{\mathsf{N1}, \mathsf{N2}\}$.

The refinement technique that we propose consists of performing a breadth-first traversal of the structures until the first symbolic value in the search is found. Let us denote by $posBFS(\mathsf{N}, S)$ the position of an arbitrary node N in the breadth-first traversal of the structure S, prior to the first symbolic node. We know that identifiers in N's label that differ from $posBFS(\mathsf{N}, S)$ may be removed.

Algorithm 3 shows the refinement algorithm. It returns a boolean value, indicating whether the (possibly) refined structure is still valid, or became redundant due to the refinement. Lines 22-25 show how the labels for concrete nodes, prior to the first symbolic node in breadth-first traversal, are set. Lines 19-21 show that if the current label of a node does not contain its only valid position, then the structure is spurious and can be removed. Algorithm 4 shows the pseudocode for the RBLI algorithm. Notice how this algorithm builds over BLI, by enabling the systematic path exploration to

backtrack when the current partially symbolic structure is found to be spurious.

Algorithm 3. The heap refinement algorithm.

```
1  boolean refineHeap(Heap h)
2    Set⟨HeapNode⟩ roots = h.getRoots();
3    if (not roots.isEmpty()) then
4      Queue⟨HeapNode⟩ pending = new List⟨HeapNode⟩();
5      Set⟨HeapNode⟩ visited = new Set⟨HeapNode⟩();
6      for (HeapNode hn : roots) do
7        pending.add(r);
8        visited.add(r);
9      end
10     int currIndex = 1;
11     boolean foundSymbolic = false;
12     while (not pending.isEmpty() and not foundSymbolic) do
13       HeapNode hn = pending.remove();
14       if (isSymbolic(hn)) then
15         foundSymbolic = true;
16         return true;
17       else
18         if (not isNull(hn)) then
19           if (not h.getLabel(hn).contains(currIndex)) then
20             return false;
21           end
22           LabelSet ls = new LabelSet();
23           ls.add(currIndex);
24           h.setLabel(hn, ls);
25           currIndex++;
26           for (Field f : hn.getFields()) do
27             HeapNode n = hn.getFieldValue(hn, f);
28             if (isSymbolic(n) or visited.add(n)) then
29                 pending.add(n);
30             end
31           end
32         end
33       end
34     end
35   end
36   return true;
37 end
```

Algorithm 4. Pseudocode for the Refined Bounded Lazy Initialization (RBLI) algorithm.

```
if (f is uninitialized) then
  if (f is reference field of type T) then
    nondeterministically initialize f to
    1. null
    2. a new object n of class T (with uninitialized fields)
                    and label(n) := label(this,f)
    3. an object x created during a prior
                    initialization of a field of type T
                    such that label(this).intersects(label(x))
    if (method precondition is violated || !refineHeap
     (currentHeap)) then
        backtrack();
    end
  end
  if (f is primitive field) then
    initialize f to a new symbolic value of appropriate type
  end
end
```

The following theorem shows that RBLI is sound and complete with respect to LI.

Theorem 2. *RBLI is sound and complete with respect to LI, i.e., a valid (with respect to the concrete imperative precondition) structure is produced by RBLI if and only if it is produced by LI.*

Proof. Soundness is straightforward. Since RBLI differs from BLI in that the former incorporates a process for bound refinement, RBLI cannot produce instances that are not produced by BLI. Since BLI is sound with respect to LI, RBLI is sound with respect to LI as well.

The proof of completeness of RBLI with respect to LI is based on the fact that every concrete node N in a symbolic structure S has as a label a set that contains all the identifiers that can be assigned to N along BFS traversals of any fully concrete extension of S. Since our approach is based on the assumption that correct field bounds are used, we can assume that RBLI executes on symbolic structures that satisfy the above condition, to show that RBLI does not prune valid structures.

Let S_0 be a symbolic structure. Let N_i be the concrete node in S_0 in position i in the BFS traversal of S_0. Let LS_i be the label set for node N_i, and let RLS_i be the refined label set for N_i produced by RBLI. Let us suppose that there exists a valid fully concrete structure S that extends S_0 that will not be generated due to discarding S_0. Also, let us suppose $\mathsf{N}_i \notin RLS_i$ (thus pruning the valid structure S). Certainly, $\mathsf{N}_i \in LS_i$ due to the completeness of BLI with respect to LI. Then, since RBLI differs from BLI only in bound refinement, N_i must have been removed from RLS_i during refinement. Recall that the refinement process stops as soon as a symbolic node is reached. Thus, the fact label set LS_i was refined implies N_i must occur in a fully concrete prefix of the BFS traversal of structure S_0.

Let $posBFS(\mathsf{N}_i, S)$ be the index of node N_i in the BFS traversal of structure S.

- If $i < posBFS(\mathsf{N}_i, S)$, then node N_i appears in a smaller BFS traversal position in S_0 than its final BFS position in S. Since what differentiates S_0 from S is the concretization of symbolic values, node N_i must appear in a previous BFS position in S_0 due to the concretization of a symbolic value in S_0 that occurs before N_i in the breadth-first traversal. But then, since there is a symbolic value prior to N_i in the BFS traversal of S, LS_i is not refinable, i.e., $RLS_i = LS_i$. This contradicts the fact that $\mathsf{N}_i \in LI_i$ and $\mathsf{N}_i \notin RLS_i$.
- If $i > posBFS(\mathsf{N}_i, S)$, then we have a node in a structure whose label is strictly greater than its BFS position, contradicting the symmetry-breaking predicates, which force a breadth-first canonical reference assignment for structures. □

The completeness of RBLI (as well as the completeness of BLI) with respect to LI indicates that the technique will only prune redundant structures. As mentioned before, this means not just that the systematic exploration of feasible paths will be more efficient, but also that, if the resulting path conditions are solved to build test suites, the suites

```
public static void main(String[] args) {
    TreeSet X = new TreeSet();
    X = (TreeSet) Debug.makeSymbolicRef("X", X);
    try {
        if (X != null)
            X.bfsTraverse();
    } catch (Throwable t) {}
}
```

Fig. 10. A "main" method driving the SPF analysis of method bfsTraverse.

obtained by RBLI and BLI may be smaller than those obtained by LI, resulting in fewer test cases. Thanks to completeness, the test cases that will not be part of suites built using BLI or RBLI, or any other of the techniques to be introduced in this paper, will be redundant with respect to the concrete precondition of the method under analysis, and therefore also spurious. As it will be shown in Section 5, bound refinement often provides most of the observed speedups of our techniques over LI.

4 BLISS AND BLISSDB

Bounded lazy initialization with SAT support is the second main contribution of this paper. BLISS extends RBLI with a mechanism for pruning partially symbolic structures as soon as their concretization is determined to be *infeasible*, i.e., they cannot possibly be concretized to become a valid fully concrete structure. This is achieved by searching for concretizations of partially symbolic structures using the corresponding preconditions (e.g., data structure class invariants), and resorting to SAT solving. When no concretization exists for a given partially symbolic structure, it can be safely discarded in the symbolic execution process. Since these SAT solver invocations are costly, we introduce an optimization of BLISS called BLISSDB in which SAT call verdicts are cached in a database and reused across different analyses. This section introduces BLISS, and then describes the BLISSDB optimization.

BLISS prunes redundant partially symbolic structures that could not be pruned by the previously introduced techniques. To this end, BLISS conjoins the declarative class invariant with a propositional description of the concrete part of the structure; since no constraints are imposed on the symbolic parts of the structure, a SAT-solver call allows us to determine whether there is a way to concretize the parts that still remain symbolic in order to obtain a fully concrete structure. This high-level intuition will be made precise in this section.

To illustrate BLISS, let us consider a somewhat more complex example than the previously shown one (binary trees). Fig. 10 shows a `main` program used as a driver for the symbolic execution of a breadth-first traversal of a red-black tree. Red-black trees are balanced binary search trees. They are used as the implementation of class TreeSet in package java.util.collections, and satisfy the following constraints, which constitute their class invariant:

- *rbt1:* the tree is a binary search tree,
- *rbt2:* each node has a color, which can be red or black,
- *rbt3:* the tree root is black,
- *rbt4:* no two consecutive nodes in a path can be red, and
- *rbt5:* all the paths from the root to a leaf contain the same number of black-colored nodes.

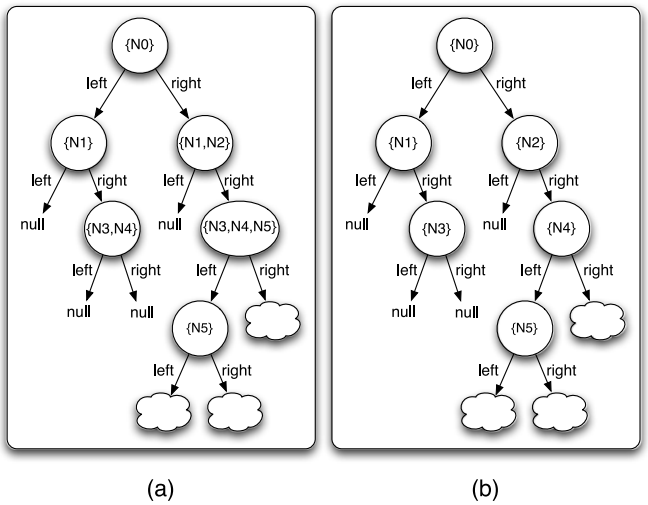


Fig. 11. A partially symbolic red-black tree that is considered valid by LI and BLI (a), and its RBLI-refined version (b), also considered valid by RBLI, but pruned by BLISS.

Consider the partially symbolic red-black tree depicted in Fig. 11a, which is generated during the symbolic execution of method bfsTraverse. Let us discuss why this tree cannot be colored in a way that satisfies the class invariant, which is reasonable to assume would be part of the precondition for bfsTraverse. First, condition *rbt1* forces the root node to be black. The coloring of root.left deserves some analysis. If root.left is red, then root.left.right must be red too in order to satisfy condition *rbt5*. But this leads to a violation of condition *rbt4*. Therefore, root.left must be black. However, since all the paths from the root to a leaf node must contain the same number of black nodes, they must all have exactly two black nodes (one of them being the root node). It is then impossible to give a valid coloring to the right subtree (the reader should convince him/herself of this by looking for a valid coloring).

Even though the tree in Fig. 11a is redundant, due to our discussion in Section 2.1.1, this tree would be generated during the execution of LI. Moreover, the node labeling, computed according to the TACO bounds for red-black trees with up to six nodes, shows that this tree will also be generated by BLI. It will even be generated by RBLI, as illustrated by Fig. 11b, which shows the same tree after refinement has been applied to the node labels. Unlike LI, BLI and RBLI, BLISS would recognize the infeasibility of Fig. 11a via a satisfiability check on the structure, pruning it and consequently not considering it, nor any of its extensions.

To perform BLISS satisfiability checks we use a *declarative* precondition for the method under analysis, which must be provided by the user. For instance, for method bfsTraverse, the precondition would be the red-black tree class invariant applied to the method's parameter, expressed in a language such as JML [4]. Fig. 12 presents a fragment of such an invariant.

In order to perform the above described satisfiability checks, we automatically translate the method's declarative precondition to a SAT-solving problem using the TACO tool. We combine the propositional translation of the declarative precondition with a characterization of the partially symbolic structure whose feasibility we want to check. To this

```
/*@
  @ invariant root!=null ==> root.color== BLACK;
  @ invariant
  @   (\forall Node n;
  @    \reach(root, Node, left).has(n);
  @       ((n.color == RED && n.left != null) ==>
  @          n.left.color == BLACK) &&
  @       (\forall Node x;
  @        \reach(n.left, Node, left).has(x);
                x.key < n.key) &&
  @       (\forall Node x;
  @        \reach(n.right, Node, left).has(x);
               x.key > n.key) &&
  @  ...
  @*/
```

Fig. 12. A fragment of the JML class invariant for TreeSet.

end, BLISS uses a mapping called *pvars* (for "*p*ropositional *var*iables") whose key set is made of triples of the form:

$$\langle sourceHeapNode, field, targetHeapNode \rangle$$

and whose values are variable numbers in the DIMACS [25] CNF encoding of the propositional formula that results from the translation of the declarative precondition. Intuitively,

$$pvars.\texttt{get}(\langle s, \mathsf{f}, t \rangle) = v$$

means that v is the variable that encodes the fact that field f maps the source heap node s to the target heap node t. For efficiency reasons to be discussed in Section 5.3, the analysis is simplified if we consider nodes s and t whose label sets are singletons. Note that this naturally happens in the fully concrete prefixes of the BFS traversal of a heap, after RBLI has been applied. Therefore, we apply BLISS after RBLI, in order to benefit from these fully concrete prefixes. BLISS makes use of *SAT-solving under assumptions*, a common SAT-solver feature that allows one to call the solver repeatedly on the same CNF instance, each time passing as parameter a different set of *assumptions* (literals that will be assumed to hold during the analysis). BLISS performs a BFS traversal of the heap and generates the assumptions to be used during that analysis. Algorithm 5 presents BLISS's traversal process. Note that, at its core, it collects as assumptions the fully concrete prefixes of the BFS traversal of the partially symbolic structure. For our refined sample structure in Fig. 11b, the SAT-solver would solve the propositional translation of the declarative class invariant under the following assumptions:

$$pvars.\texttt{get}(\langle N_0, left, N_1 \rangle), \quad pvars.\texttt{get}(\langle N_0, right, N_2 \rangle),$$
$$pvars.\texttt{get}(\langle N_1, left, null \rangle), \quad pvars.\texttt{get}(\langle N_1, right, N_3 \rangle),$$
$$pvars.\texttt{get}(\langle N_2, left, null \rangle), \quad pvars.\texttt{get}(\langle N_2, right, N_4 \rangle),$$
$$pvars.\texttt{get}(\langle N_3, left, null \rangle), \quad pvars.\texttt{get}(\langle N_3, right, null \rangle),$$
$$pvars.\texttt{get}(\langle N_4, left, N_5 \rangle).$$

Algorithm 6 shows the pseudocode of the BLISS algorithm, where the use of the `processHeapWithSolver` routine is indicated. Notice again that this algorithm builds over RBLI, adding a new condition under which the systematic path exploration is forced to backtrack, namely, when the current partially symbolic heap is found to be spurious by a SAT query. Also, `processHeapWithSolver` is executed *after* `refineHeap`, therefore operating on the already refined label sets of the nodes in the partially symbolic heap.

Algorithm 5. Assumptions computation in BLISS.

```
1  boolean processHeapWithSolver(Heap h)
2    Node root = getRoot(h);
3    if (not isSymbolic(root) and not isNull(root)) then
4      Queue⟨Node⟩ pending = new LinkedList⟨Node⟩();
5      HashSet⟨Node⟩ visited = new HashSet⟨Node⟩;
6      HashSet⟨Integer⟩ assumptions = new HashSet⟨Integer⟩;
7      pending.add(root);
8      visited.add(root);
9      While (not pending.isEmpty()) do
10       Node src = pending.remove();
11       if (not isSymbolic(src) and not isNull(src)) then
12         for (String fn : classFieldNames) do
13           Node target = pointsThroughField(src, fn);
14           assumptions.add(pvars(src, fn, target));
15           if (visited.add(target)) then
16             pending.add(target)
17           end
18         end
19       end
20     end
21     boolean verdict = solver.isSatisfiable(assumptions);
22     return verdict;
23   end
24   return true;
25 end
```

Algorithm 6. Pseudocode of the BLISS algorithm.

```
if (f is uninitialized) then
  if (f is reference field of type T) then
    nondeterministically initialize f to
    1. null
    2. a new object n of class T (with uninitialized fields)
                    and label(n) := label(this,f)
    3. an object x created during a prior
                    initialization of a field of type T
                    such that label(this).intersects(label(x))
    if (method precondition is violated || !refineHeap(currentHeap) ||
        !processHeapWithSolver(currentHeap)) then
      backtrack();
    end
  end
  if (f is primitive field) then
    initialize f to a new symbolic value of appropriate type
  end
end
```

BLISS satisfiability checks serve, in many situations, the same purpose that preconditions/class invariants on partially symbolic structures serve during LI, i.e., to rule out partially symbolic structures that cannot be extended to a valid concrete structure. However, BLISS improves upon preconditions under various aspects. First, preconditions must be generalized to deal with symbolic structures. This task has to be manually carried out by the engineer, who has to attempt to algorithmically decide if a partially symbolic structure is concretizable from the already concretized part of the structure. Typically, this approach is limited in the way it refers to symbolic portions of the partially symbolic structure, and is

dependent on how well the engineer is able to "generalize" the concrete precondition/class invariant to symbolic structures. Second, preconditions or imperative class invariants do not take the scope—that is, the number of available nodes, loop iterations, and so on—into consideration. Satisfiability checks based on the translation of declarative class invariants, on the other hand, can predicate on symbolic portions of the structure straightforwardly (essentially, via existential quantification), and are able to draw conclusions based on the scope, which is a necessary part of every satisfiability check, as in the previous example.

On the other hand, using BLISS needs some additional effort in comparison with LI and our previously introduced extensions: it requires the engineer to provide a *declarative invariant*. We shall elaborate on this fact in Section 5.4.

As the following theorem states, BLISS is sound and complete (assuming equivalence of the declarative and procedural invariants) with respect to LI. When we refer to the equivalence with the procedural class invariant we mean with respect to the one that operates on fully concrete structures, and not the hybrid one. The latter is a more general and weaker version of the former.

Theorem 3. *BLISS is sound and complete with respect to LI, i.e., a valid structure is produced by BLISS if and only if it is produced by LI, provided that, for the class under analysis, the declarative class invariant used by BLISS is equivalent to the imperative class invariant on which the hybrid invariant used by LI is based.*

Proof. BLISS extends RBLI with satisfiability checks on partially symbolic structures. Notice that these checks are only used for pruning, so BLISS cannot generate any structure that RBLI would not generate. Then, since RBLI is sound with respect to LI, BLISS is also sound with respect to LI.

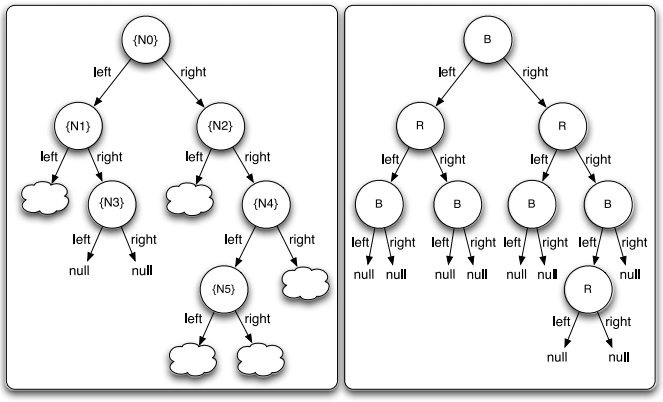
Now let us prove completeness. Let S be a valid structure generated by LI within scope k, but rejected by BLISS for the same scope. It certainly is not rejected due to bound refinement, since RBLI is complete with respect to LI. Then, its generation by BLISS has been prevented due to satisfiability checks. That is, there must exist a partially symbolic structure S' such that S extends S', and BLISS found S' to be redundant. Since the encoding of declarative class invariants and partially symbolic structures employed by BLISS is sound and complete with respect to bounded verification (cf. [11]), the encoding of S' being infeasible implies that there is no concrete structure within scope k that extends S' and satisfies the declarative class invariant. But, since S extends S', it is within scope k and satisfies the concrete imperative class invariant, it must be the case that the declarative and imperative invariants are not equivalent, contradicting our hypothesis. □

Satisfiability checks usually take substantially more time than concrete executions of the kind that LI performs when executing an imperative hybrid repOK or precondition on partially symbolic structures. Thus, BLISS SAT checks will be worthwhile only if the number of structures pruned thanks to these checks is greater than those pruned by hybrid precondition checking in LI, so that the cost of SAT checks pays off with respect to considering a significantly larger number of structures (those generated if only preconditions on symbolic structures are employed). The level of pruning provided by BLISS depends on the strength of the corresponding class invariant, and on the method under analysis and how it traverses the structure. We will show in Section 5 that, in our case studies, BLISS improves the analysis time in 25 (out of 32) methods with respect to LI. Moreover, BLISS allowed us to analyze a method, namely AvlTree.bfs, using scope 20, whereas LI runs out of memory in scope 13 (cf. Table 3).

Again, as for the case of RBLI, since BLISS helps in reducing the number of partially symbolic structures collected while systematically exploring path conditions, it provides significant advantages in automated test input generation considering test suites built by concretizing the collected partially symbolic structures. Furthermore, since BLISS is sound and complete with respect to LI, only redundant cases are dismissed by the technique, ensuring that we retain exactly the same coverage obtained by using standard SPF with LI. As our experiments show (cf. Table 10), in some cases we achieve reductions of up to 99.8 percent on the number of partially symbolic structures produced using LI. As we will show in the experimental evaluation section, among the techniques introduced in this paper, BLISS is the one responsible for most of the reductions in the number of partially symbolic structures obtained during systematic path exploration.

Unlike LI and our previously introduced techniques BLI and RBLI, BLISS introduces additional SAT checks during SPF's symbolic execution. One might wonder what the advantage could be of using these additional SAT checks, considering the fact that SPF already uses SMT-solving to prune the search space by solving path conditions. However, note that, while the SMT checks performed internally by SPF have a complexity that depends on the structure of the program under analysis (since these are used to solve individual path conditions), the additional SAT checks performed by our technique are completely independent of the program structure—they only depend on the program's precondition and scope. Since preconditions/invariants on data structures usually involve quantifiers and reachability constraints, we consider that this separation of concerns positively contributes to the performance of the approach.

The above described technique requires a significant number of satisfiability checks—a priori, one per each partially symbolic structure found along the symbolic execution process. If we consider the workflow that users typically employ when performing bounded analyses of the kind offered by Symbolic PathFinder, we note that a large number of those checks can be reused. An engineer using a bounded verification tool will generally want to check a property for increasingly large scopes: he or she would begin by checking a property for some small scope, so as to ensure termination within reasonable time, and then perform checks for larger and larger scopes, in order to gain greater confidence in the validity of the property (until eventually reaching a scope where the tool needs more time or space than available). Thus, when checking a property for a scope k, we may find it



(a) (b)

Fig. 13. A partially symbolic red-black tree redundant in scope 6, and a valid concrete extension in scope 8.

useful to reuse any equivalent satisfiability checks that might already have been performed on prior analyses for scopes smaller than k.

As an example, consider again the structure in Fig. 9a. BLISS would check this structure's feasibility before extending it, when performing symbolic execution for a given scope, say, 3. Note that if instead of 3, the scope is 4 (or any other number greater than 3), the structure in Fig. 9a will still be feasible under this new scope. This is the case due to two facts. Firstly, scopes are *non strict* in our analyses, in the sense that checking a property α for a scope k corresponds to verifying whether α holds or not for all inputs whose size is *at most* k (as opposed to strict analysis, which would check the property for inputs of size exactly k). Secondly, (correct) TACO field bounds are monotonic with respect to scope incrementation, as the following theorem shows.

TABLE 1
Analysis Time and Speedup for Class TreeSet (All Techniques)

Method	Technique	S07	S08	S09	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
bfs	LI	00:05	00:17	00:58	03:37	13:43	53:43	179:10	OOM						
	BLI	00:03	00:11	00:31	02:15	09:33	39:25	OOM							
	RBLI	**00:01**	00:04	00:11	00:38	02:29	09:50	39:28	OOM						
	BLISS	**00:01**	**00:02**	**00:03**	00:06	00:12	00:29	01:13	03:05	07:45	20:02	**55:38**	OOM		
	BLISSDB	**00:01**	**00:02**	**00:03**	**00:05**	**00:10**	**00:23**	**00:56**	**02:15**	**05:13**	**12:58**	OOM			
	Speedup	5X	8X	19X	43X	82X	140X	188X	∞	∞	∞	∞			
dfs	LI	00:02	00:05	00:19	01:13	05:44	OOM								
	BLI	00:01	**00:02**	00:06	00:26	02:16	OOM								
	RBLI	**00:00**	**00:02**	**00:04**	**00:16**	**01:17**	**08:03**	**27:38**	OOM						
	BLISS	00:01	00:04	00:11	00:58	05:06	23:21	135:14	TO						
	BLISSDB	00:01	00:04	00:12	01:10	05:24	22:41	103:26	485:27	TO					
	Speedup	2X	2X	4X	4X	4X	∞	∞	∞						
repOK	LI	03:34	22:46	150:44	OOM										
	BLI	01:58	15:58	78:49	OOM										
	RBLI	00:55	06:38	24:10	194:17	OOM									
	BLISS	**00:17**	**00:49**	01:57	05:39	22:21	**96:36**	OOM							
	BLISSDB	**00:17**	**00:49**	**01:56**	**05:30**	**22:11**	98:37	OOM							
	Speedup	12X	27X	77X	∞	∞	∞								
contains	LI	00:05	00:12	00:29	01:07	02:39	06:11	14:35	32:54	75:11	174:47	OOM			
	BLI	**00:01**	**00:02**	**00:02**	**00:05**	**00:12**	**00:28**	**01:06**	**02:37**	**06:02**	**13:57**	**32:37**	OOM		
	RBLI	**00:01**	**00:02**	**00:02**	**00:05**	**00:12**	**00:28**	01:08	02:38	06:07	14:10	33:03	**74:17**	OOM	
	BLISS	**00:01**	**00:02**	**00:02**	**00:05**	00:13	00:31	01:15	02:54	06:52	16:21	38:15	88:54	204:01	OOM
	BLISSDB	**00:01**	**00:02**	**00:02**	**00:05**	00:13	00:30	01:12	02:49	06:32	15:21	35:47	83:00	190:06	OOM
	Speedup	5X	12X	14X	13X	13X	13X	13X	12X	12X	12X	∞	∞	∞	
add	LI	03:50	23:47	150:57	OOM										
	BLI	02:17	17:02	81:03	OOM										
	RBLI	01:12	07:19	25:35	200:55	OOM									
	BLISS	**00:33**	**01:30**	03:32	**09:13**	31:01	**118:22**	OOM							
	BLISSDB	00:34	01:31	**03:30**	09:04	**30:40**	119:10	OOM							
	Speedup	6X	15X	43X	∞	∞	∞								
remove	LI	01:55	07:35	30:11	112:00	417:00	OOM								
	BLI	00:54	04:22	11:12	**53:17**	**235:39**	OOM								
	RBLI	**00:53**	**04:18**	**10:56**	53:33	237:13	OOM								
	BLISS	00:55	04:34	11:57	59:04	OOM									
	BLISSDB	00:54	04:24	11:02	OOM	OOM									
	Speedup	2X	1X	2X	2X	1X									

TABLE 2
Analysis Time and Speedup for Class TreeMap (All Techniques)

Method	Technique	S01	S02	S03	S04	S05	S06	S07	S08	S09	S10	S11	S12	S13
trace_L1	LI	**00:00**	**00:00**	00:01	00:04	00:16	00:57	03:29	12:44	47:03	173:20	OOM		
	BLI	**00:00**	**00:00**	**00:00**	**00:02**	**00:03**	**00:17**	01:30	06:43	15:58	76:21	OOM		
	RBLI	**00:00**	**00:00**	**00:00**	**00:02**	**00:03**	00:18	**01:25**	**06:31**	15:40	**74:34**	**323:47**	OOM	
	BLISS	**00:00**	**00:00**	**00:00**	**00:02**	**00:03**	**00:17**	01:28	06:49	16:56	81:28	373:01	OOM	
	BLISSDB	**00:00**	**00:00**	**00:00**	**00:02**	**00:03**	00:18	01:26	06:41	**15:37**	76:16	342:54	OOM	
	Speedup		1X	1X	2X	5X	3X	2X	1X	3X	2X	∞		
trace_L2	LI	**00:00**	**00:14**	01:25	07:58	41:30	205:21							
	BLI	00:02	00:15	**00:29**	04:03	07:50	72:00	OOM						
	RBLI	00:02	**00:14**	**00:29**	**03:23**	**07:11**	60:33	OOM						
	BLISS	00:02	**00:14**	**00:29**	03:24	07:15	57:32	OOM						
	BLISSDB	00:02	00:15	00:30	03:28	07:22	**57:08**	OOM						
	Speedup	0.5X	1X	2X	2X	5X	3X							
trace_L3	LI	**00:49**	08:04	73:13	OOM									
	BLI	00:50	08:03	21:46	OOM									
	RBLI	**00:49**	08:11	**21:43**	OOM									
	BLISS	**00:49**	**08:02**	21:46	215:34	OOM								
	BLISSDB	**00:49**	08:07	21:52	216:03	OOM								
	Speedup	1X	1X	3X	∞									

Theorem 4. *Let C be a class and f a field in it, of type C'. Let U_{f}^k and U_{f}^{k+1} be TACO field bounds for f for scopes k and $k+1$, respectively. Then, U_{f}^k is contained in U_{f}^{k+1}.*

Proof. Let t be a tuple in U_{f}^k. Since TACO bounds are the tightest for a corresponding class and scope, there must exist a valid structure c of class C within scope k, such that $\langle c, c.\mathsf{f}\rangle = t$. Since c is a valid structure within scope k, it is also a valid structure within scope $k+1$. Therefore, by the correctness of TACO bounds, $\langle c, c.\mathsf{f}\rangle$ must belong to U_{f}^{k+1}, i.e., $t \in U_{\mathsf{f}}^{k+1}$. □

The above observation involves only the satisfiable cases: whenever a symbolic structure is found to be feasible for scope k, it will also be feasible for scopes greater than k. However, the same situation does not hold for unsatisfiable cases. A symbolic structure may be redundant due to scope restrictions, but its concretization may become feasible if larger scopes were used. For instance, Fig. 13a is redundant as a partially symbolic red-black tree when the scope is 6. This is because there are already six concrete nodes; hence, all possible extensions concretize symbolic references to the null value. This leads to a tree that is a concretization of the tree in Fig. 11a, which we already discussed was not concretizable as a red-black tree. Fig. 13b, on the other hand, depicts an appropriately-colored concrete extension of the tree in Fig. 13a. This means that the tree in Fig. 13a is feasible in scope 8. Therefore, infeasibility in a given scope does not necessarily promote to larger scopes. The BLISSDB technique consists then of an optimization to BLISS, that takes advantage of previous SAT computations by *caching* the results for satisfiable cases. Algorithm 7 shows BLISSDB's traversal process. Essentially, BLISSDB is the same as Algorithm 6, but instead of calling `processHeapWithSolver`, it calls `processHeapWithSolverDB`, which caches satisfiable results.

In Section 5 we will discuss experimental results. It is our experience that depending on the SMT and SAT solvers being used, BLISSDB can produce a 2X speedup over BLISS.

Algorithm 7. Assumptions computation in BLISSDB.

```
1  boolean processHeapWithSolverDB(Heap h)
2    Node root = getRoot(h);
3    if (not isSymbolic(root) and not isNull(root)) then
4      Queue⟨Node⟩ pending = new LinkedList⟨Node⟩();
5      HashSet⟨Node⟩ visited = new HashSet⟨Node⟩;
6      HashSet⟨Integer⟩ assumptions = new HashSet⟨Integer⟩;
7      pending.add(root);
8      visited.add(root);
9      while (not pending.isEmpty()) do
10       Node src = pending.remove();
11       if (not isSymbolic(src) and not isNull(src)) then
12         for (String fn : classFieldNames) do
13           Node target = pointsThroughField(src, fn);
14           assumptions.add(pvars(src, fn, target));
15           if (visited.add(target)) then
16             pending.add(target);
17           end
18         end
19       end
20     end
21     boolean verdict;
22     if (foundSatisfiable.get(assumptions)) then
23       verdict = true;
24     else
25       verdict = solver.isSatisfiable(assumptions);
26       if (verdict) then
27         foundSatisfiable.add(assumptions);
28       end
29     end
30     return verdict;
31   end
32   return true;
33 end
```

TABLE 3
Analysis Time and Speedup for Class AvlTree (All Techniques)

Method	Technique	S07	S08	S09	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
bfs	LI	00:05	00:16	00:55	03:22	12:43	49:00	OOM							
	BLI	00:01	00:07	00:16	00:35	01:16	09:52	60:03	OOM						
	RBLI	**00:00**	**00:01**	00:04	00:09	00:14	00:44	03:23	15:54	73:22	OOM				
	BLISS	**00:00**	**00:01**	**00:02**	**00:03**	**00:05**	00:10	00:22	00:50	01:47	03:56	08:08	16:12	29:25	OOM
	BLISSDB	**00:00**	**00:01**	**00:02**	**00:03**	**00:05**	**00:09**	**00:20**	**00:42**	**01:26**	**02:56**	**05:36**	**10:01**	**17:09**	**36:33**
	Speedup	5X	16X	27X	37X	152X	326X	∞	∞	∞	∞	∞	∞	∞	∞
dfs	LI	00:03	00:10	00:34	02:03	08:23	OOM								
	BLI	00:01	00:04	00:07	00:14	00:27	03:53	22:30	OOM						
	RBLI	**00:00**	**00:01**	**00:03**	00:06	00:13	**01:18**	**08:30**	OOM						
	BLISS	**00:00**	**00:01**	**00:03**	**00:05**	**00:10**	01:38	14:04	**90:10**	**476:05**	TO				
	BLISSDB	00:01	00:02	**00:03**	00:06	**00:10**	03:18	20:53	111:53	561:25					
	Speedup	3X	10X	11X	24X	50X	∞	∞	∞	∞					
repOK	LI	01:18	03:49	10:51	29:50	82:10	223:34	OOM							
	BLI	01:03	03:44	10:18	27:19	71:33	219:25	OOM							
	RBLI	00:33	02:01	06:32	17:42	37:45	110:42	OOM							
	BLISS	00:29	01:25	03:28	**07:34**	**16:23**	38:52	**103:02**	264:35	OOM					
	BLISSDB	**00:28**	**01:24**	**03:24**	07:36	16:25	**38:26**	104:18	**258:11**	OOM					
	Speedup	2X	2X	3X	3X	4X	5X	∞	∞						
contains	LI	00:05	00:12	00:29	01:07	02:42	06:16	14:26	32:46	74:51	OOM				
	BLI	**00:00**	**00:01**	**00:01**	**00:01**	**00:01**	**00:02**	**00:05**	**00:12**	**00:28**	01:08	**01:05**	**01:06**	01:07	**02:36**
	RBLI	**00:00**	**00:01**	**00:01**	**00:01**	**00:01**	**00:02**	**00:05**	**00:12**	**00:28**	**01:06**	01:06	01:07	**01:06**	02:39
	BLISS	**00:00**	**00:01**	**00:01**	**00:01**	**00:01**	**00:02**	**00:05**	00:13	00:30	01:11	01:13	01:13	01:14	03:02
	BLISSDB	**00:00**	**00:01**	**00:01**	**00:01**	**00:01**	**00:02**	**00:05**	00:13	00:29	01:11	01:13	01:07	01:08	02:54
	Speedup	5X	12X	29X	67X	162X	188X	173X	163X	160X	∞	∞	∞	∞	∞
insert	LI	18:28	91:16	OOM											
	BLI	02:03	23:45	30:43	30:36	30:45	OOM								
	RBLI	01:44	**18:37**	**26:14**	26:15	**26:07**	OOM								
	BLISS	**01:42**	19:00	26:30	26:16	26:32	OOM								
	BLISSDB	01:44	18:57	26:34	**26:06**	26:27	OOM								
	Speedup	10X	3X	∞	∞	∞									
remove	LI	117:33	OOM												
	BLI	25:18	**443:50**	OOM											
	RBLI	18:42	OOM												
	BLISS	**18:01**	OOM												
	BLISSDB	18:02	OOM												
	Speedup	6X	∞												

5 Evaluation

Before getting into the description of the experimental results, note that each of the presented techniques builds over the previous ones: BLI extends LI, RBLI extends BLI, BLISS extends RBLI, and BLISSDB extends BLISS. Since all the techniques are sound and complete with respect to LI, only spurious paths (and the corresponding partially symbolic structures) are removed by our techniques. Also, each technique introduces further pruning over the previous one, since each one incorporates some new element (compared with the corresponding previous technique) to decide the infeasibility of a path/symbolic structure. However, it is not obvious whether these mechanisms pay off from an efficiency point of view, since a particular pruning technique could be too costly while only removing a small number of spurious paths/symbolic structures. In this section we evaluate this issue, i.e., whether the techniques introduced in this paper are worthwhile.

The BLI, RBLI, BLISS and BLISSDB algorithms described in the previous sections were incorporated into the standard distribution of Symbolic PathFinder and compared with the already-implemented LI algorithm. To assess these algorithms, we used a benchmark consisting of 32 methods from the following data structures:

- LinkedList: An implementation of the AbstractList abstract datatype based on circular doubly-linked lists, taken from the java.util package. We consider methods repOK (which checks that the structure is actually a well-formed doubly-linked circular list), add (which appends the given element at the end of the list), contains (which returns true if the list contains a given element), and remove (which removes the element stored in a certain position in the list). Note that this class is implemented using a cyclic structure, which shows the suitability of the techniques for such structures as well.

TABLE 4
Analysis Time and Speedup for Class BinomialHeap (All Techniques, Methods bfs, dfs and repOK)

Method	Technique	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
bfs	LI	00:59	03:33	12:52	48:40	171:59	TO					
	BLI	00:18	00:59	03:11	11:38	42:49	140:52	520:12	TO			
	RBLI	00:01	00:02	00:05	00:13	00:39	01:20	01:36	02:11	02:40	03:40	07:31
	BLISS	**00:00**	**00:00**	00:01	**00:01**	**00:01**	**00:01**	**00:01**	00:02	00:02	00:03	00:03
	BLISSDB	**00:00**	**00:00**	**00:00**	**00:01**	**00:01**	**00:01**	**00:01**	**00:01**	**00:01**	**00:01**	**00:02**
	Speedup	59X	213X	772X	2,920X	10,319X	>36,000X	≫36,000X	⋙36,000	⋙36,000	⋙⋙36,000	⋙⋙18,000
dfs	LI	00:49	02:44	09:07	30:53	103:15	338:52	TO				
	BLI	00:08	00:15	01:00	02:01	08:59	25:45	50:12	111:33	247:05	476:14	TO
	RBLI	**00:05**	**00:12**	**00:34**	**01:40**	**04:47**	**13:55**	**45:23**	**97:10**	**190:02**	**374:59**	TO
	BLISS	00:08	00:23	01:06	03:40	10:18	32:07	98:17	224:52	450:17	TO	
	BLISSDB	00:10	00:29	01:22	03:53	09:43	28:51	81:16	167:02	486:43	TO	
	Speedup	9X	13X	16X	18X	21X	24X	>13X	≫6X	⋙3X	⋙1X	
repOK	LI	00:25	00:46	01:24	02:34	04:44	08:28	15:13	27:21	50:08	87:44	156:38
	BLI	00:10	00:17	00:24	00:41	01:04	01:36	02:02	02:30	03:04	03:45	06:23
	RBLI	**00:09**	**00:15**	00:21	00:36	01:01	**01:31**	01:55	02:19	02:46	03:25	05:31
	BLISS	**00:09**	**00:15**	00:21	00:36	**00:58**	01:33	01:56	02:23	02:53	03:28	05:45
	BLISSDB	**00:09**	**00:15**	**00:20**	**00:35**	**00:58**	01:32	**01:51**	**02:15**	**02:41**	**03:11**	**05:30**
	Speedup	2X	3X	4X	4X	4X	5X	8X	12X	18X	27X	28X
extractMin (bug find)	LI	00:31	00:58	01:38	00:34	00:50	01:49	02:41	04:15	07:00	12:22	20:53
	BLI	00:16	00:29	00:38	00:18	00:19	00:55	00:58	01:01	01:03	01:34	01:52
	RBLI	**00:15**	**00:27**	**00:35**	00:17	00:19	**00:53**	01:57	00:58	01:01	**01:31**	01:46
	BLISS	**00:15**	00:28	**00:35**	00:16	00:17	00:55	**00:56**	**00:57**	**00:59**	**01:31**	01:44
	BLISSDB	**00:15**	**00:27**	**00:35**	**00:16**	**00:16**	00:54	**00:56**	**00:57**	**00:59**	**01:31**	**01:42**
	Speedup	2X	2X	2X	2X	3X	2X	2X	4X	7X	8X	12X

- BinTree: An implementation of binary trees. We consider methods bfs (for breadth-first search traversal of a tree), dfs (for depth-first search traversal of a tree), repOK (which checks that the structure is actually a tree), and count (which counts the number of nodes in a tree).
- TreeSet: An implementation of the Set abstract datatype based on red-black trees, taken from the java.util package. We consider methods for breadth-first search traversal (bfs), depth-first search traversal (dfs), repOK (which tests whether the structure is a valid TreeSet), method contains, which searches a TreeSet for a given element, and methods add and remove, which insert and remove elements, respectively.
- TreeMap: A red-black-tree-based implementation of the SortedMap abstract datatype, taken from package java.util, and used in [16]. The class includes methods containsKey, print (which traverses the underlying red-black tree), put and remove. Unlike TreeSet, where Symbolic PathFinder is used to analyze isolated methods, for this class we follow the procedure adopted in [16], which consists in analyzing sequences of method invocations of increasing length.
- AvlTree: An AVL-tree-based implementation of the Map abstract datatype, used first as a case study in [6], and also used in [10], [11]. This implementation includes methods for bfs and dfs traversals, as well as contains, insert and delete. An appropriate repOK method characterizing valid AVL trees is included as well.
- BinomialHeap: This class is a binomial heap implementation of the Heap abstract datatype, and is one of the case studies discussed in [16]. It includes methods for bfs and dfs traversals, a corresponding repOK, and methods insert, extractMin and remove. We analyze these methods in isolation. Additionally, as in [16], we also consider sequences of method invocations of increasing length. These sequences include invocations of methods insert, findMinimum, extractMin, delete and decreaseKey. Method extractMin contains a nontrivial, real-world bug, which was found in [10]; we will use this method both as a driver guiding the execution of BLISS, and also to determine whether BLISS can improve the analysis time required to find that bug using LI.

The analysis consisted both in systematically exploring program paths for the above structures and methods, and collecting the corresponding partially symbolic structures that would need to be concretized via SMT-solving to build test suites. Notice that since our techniques are sound and complete with respect to LI, although the sets of structures collected with different techniques may differ in size, the corresponding test suites will be the same for all techniques (our techniques only remove redundant cases). Analyzing our techniques both on heavily constrained structures (such as red-black trees, where tight bounds are smaller) and on less constrained structures (such as binary trees or linked lists) is relevant for a number of reasons. First, the number of partially initialized structures generated using bounded lazy initialization and the extensions presented in this paper strongly depend on the cardinality of the field bounds.

TABLE 5
Analysis Time and Speedup for Class BinomialHeap (All Techniques, Remaining Methods)

Method	Technique	S01	S02	S03	S04	S05	S06	S07	S08	S09	S10	S11	S12
insert	LI	**00:00**	**00:00**	00:01	00:10	01:09	07:03	41:36	243:03	TO			
	BLI	**00:00**	**00:00**	**00:00**	00:02	00:19	02:20	16:16	21:27	22:58	175:17	178:14	OOM
	RBLI	**00:00**	**00:00**	**00:00**	00:02	00:16	02:20	16:32	21:49	23:13	172:55	175:57	OOM
	BLISS	**00:00**	**00:00**	**00:00**	**00:01**	**00:11**	01:43	11:28	16:37	**17:35**	135:10	136:12	OOM
	BLISSDB	**00:00**	**00:00**	**00:00**	**00:01**	**00:11**	**01:42**	**11:25**	**16:20**	17:52	**131:25**	**136:09**	OOM
	Speedup	1X	1X	1X	10X	6X	4X	3X	14X	>33X	≫4X	⋙4X	
delete	LI	**00:00**	**00:00**	00:01	00:23	07:13	129:06	OOM					
	BLI	**00:00**	**00:00**	00:01	00:13	04:50	80:44	OOM					
	RBLI	**00:00**	**00:00**	**00:00**	**00:05**	02:37	56:53	OOM					
	BLISS	**00:00**	**00:00**	**00:00**	**00:05**	**01:46**	**34:39**	500:29	OOM				
	BLISSDB	**00:00**	**00:00**	**00:00**	**00:05**	**01:47**	35:53	OOM					
	Speedup	1X	1X	1X	4X	4X	3X	∞					
extractMin	LI	**00:00**	**00:00**	**00:00**	00:03	00:44	10:20	135:08	OOM				
	BLI	**00:00**	**00:00**	**00:00**	00:01	00:23	04:59	60:16	297:53	OOM			
	RBLI	**00:00**	**00:00**	**00:00**	**00:00**	00:13	04:35	57:40	289:32	OOM			
	BLISS	**00:00**	**00:00**	**00:00**	**00:00**	**00:12**	**03:45**	**45:20**	223:15	OOM			
	BLISSDB	**00:00**	**00:00**	**00:00**	**00:00**	**00:12**	**03:45**	45:54	**222:45**	OOM			
	Speedup	1X	1X	1X	3X	3X	2X	2X	∞				
trace_L1	LI	**00:00**	**00:00**	00:04	00:49	11:39	183:29	OOM					
	BLI	**00:00**	**00:00**	00:02	00:22	07:03	112:03	OOM					
	RBLI	**00:00**	**00:00**	**00:01**	**00:10**	03:58	77:53	OOM					
	BLISS	**00:00**	**00:00**	**00:01**	**00:10**	**02:45**	50:08	OOM					
	BLISSDB	**00:00**	**00:00**	**00:01**	**00:10**	02:48	**49:50**	OOM					
	Speedup	1X	1X	4X	4X	4X	3X						
trace_L2	LI	**00:02**	00:33	09:13	174:40	OOM							
	BLI	**00:02**	**00:07**	02:58	60:29	OOM							
	RBLI	**00:02**	**00:07**	01:46	27:56	OOM							
	BLISS	**00:02**	**00:07**	**01:36**	23:43	OOM							
	BLISSDB	**00:02**	**00:07**	**01:36**	**23:37**	OOM							
	Speedup	1X	4X	5X	7X								
trace_L3	LI	**00:47**	16:26	OOM									
	BLI	**00:47**	**03:12**	136:58	OOM								
	RBLI	**00:47**	03:25	77:47	OOM								
	BLISS	**00:47**	03:16	**68:12**	OOM								
	BLISSDB	**00:47**	03:16	68:51	OOM								
	Speedup	1X	5X	∞									

Second, in the BLISS technique both the cost of performing SAT checks and the corresponding pruning depend on the strength of the class invariant. Class BinTree is particularly interesting because, unlike the other classes in our benchmark, its adapted "hybrid" class invariant can precisely determine the infeasibility of partially symbolic structures. Under these circumstances, TACO bounds will not contribute, either by producing significant analysis speedups or by reducing the number of collected partially symbolic structures for test generation, when compared with those produced by LI. This is experimentally confirmed in Tables 6 and 10.

5.1 Experimental Setup

Throughout this section times are presented following the pattern `mmm:ss`. "TO" (timeout) means failure to complete the analysis within 10 hours. "OOM" (out of memory) indicates failure to complete due to exhaustion of the 4 GB of JVM heap memory. TACO field bounds were not recomputed as part of the experiments for this paper. Instead, the databases of previously computed TACO bounds for the data structures involved in the experiments were reused. Computing field bounds, as put forward in [10], requires checking, via SAT-solving, the feasibility of each tuple in the corresponding field's semantic domain. Thus, a large number of SAT queries, which depend on the scope, must be performed. However, these checks are all independent from one another, and thus lend themselves to parallelization. Indeed, the approach proposed in [10] to compute tight field bounds uses a cluster. The paper includes the time required to compute bounds for the classes used in this paper, using 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. Such hardware is older and significantly slower than the one used in this paper (to be described in the next paragraph). The time may be significant (for instance, for red-black trees with up to 20 nodes it took 40:37, and for AVL trees with up to 20 nodes it

TABLE 6
Analysis Time and Speedup for Class BinTree (All Techniques)

Method	Technique	S05	S06	S07	S08	S09	S10	S11	S12	S13	S14
bfs	LI	**00:00**	00:01	00:02	00:05	00:15	00:42	03:51	OOM		
	BLI	**00:00**	**00:00**	**00:01**	00:03	00:11	00:43	02:46	OOM		
	RBLI	**00:00**	**00:00**	**00:01**	**00:02**	**00:06**	**00:17**	**01:00**	**03:42**	OOM	
	BLISS	**00:00**	**00:00**	**00:01**	00:03	00:10	00:38	02:44	OOM		
	BLISSDB	**00:00**	00:01	00:02	00:04	00:13	00:52	03:06	OOM		
	Speedup	1X	1X	2X	2X	2X	2X	3X	∞		
dfs	LI	**00:00**	**00:00**	**00:01**	00:04	00:11	00:57	03:44	OOM		
	BLI	**00:00**	**00:00**	**00:01**	**00:03**	00:11	00:43	02:46	OOM		
	RBLI	**00:00**	**00:00**	**00:01**	**00:03**	**00:09**	**00:25**	**01:35**	**06:20**	**26:53**	OOM
	BLISS	**00:00**	**00:00**	**00:01**	**00:03**	00:10	00:38	02:44	OOM		
	BLISSDB	**00:00**	00:01	**00:01**	00:06	00:26	01:30	08:07	OOM		
	Speedup	1X	1X	1X	1X	1X	2X	2X	∞	∞	
repOK	LI	**00:00**	**00:01**	**00:03**	00:10	00:34	02:03	07:40	OOM		
	BLI	**00:00**	**00:01**	**00:03**	00:09	00:30	01:46	06:37	**24:17**	OOM	
	RBLI	**00:00**	**00:01**	**00:03**	**00:08**	**00:24**	**01:22**	**04:48**	OOM		
	BLISS	**00:00**	**00:01**	**00:03**	00:09	00:29	01:43	06:28	OOM		
	BLISSDB	**00:00**	**00:01**	00:04	00:13	00:45	02:41	09:45	OOM		
	Speedup	1X	1X	1X	1X	1X	1X	1X	∞		
count	LI	**00:00**	**00:00**	**00:01**	00:04	00:13	00:53	03:43	OOM		
	BLI	**00:00**	**00:00**	**00:01**	**00:03**	00:10	00:27	**01:48**	**07:30**	OOM	
	RBLI	**00:00**	**00:00**	**00:01**	**00:03**	**00:08**	**00:23**	02:08	OOM		
	BLISS	**00:00**	**00:00**	**00:01**	00:04	00:13	00:45	03:58	OOM		
	BLISSDB	**00:00**	00:01	00:02	00:05	00:24	01:23	08:01	OOM		
	Speedup	1X	1X	1X	1X	1X	2X	2X	∞		

took 168:23 to compute the TACO bounds). Still, since bounds are used in the analysis of all methods in a class, and even across tools (these same bounds were used in TACO [11] and MUCHO-TACO [19] analyses), these bound computation times are amortized. For instance, red-black tree bounds for scope 20 were used in three methods in [11], in three methods in [19], and in nine methods (from classes TreeSet and TreeMap) in this paper. For each method in this paper we used the bounds along the analysis with BLI, RBLI, BLISS and BLISSDB. Therefore, these specific bounds were used 42 times. This makes the total time (40:37) contribute 00:58 to each analysis, which does not significantly alter the speedups achieved. Regarding the bound computation for class AvlTree, the TACO bounds for scope 20 were used in four methods in [11], in three methods in [19], and in six

TABLE 7
Analysis Time and Speedup for Class LinkedList (All Techniques)

Method	Technique	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
repOK	LI	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**
	BLI	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**
	RBLI	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**
	BLISS	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	00:01	00:01	00:01
	BLISSDB	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	00:01	00:01
	Speedup	1X	1X	1X	1X	1X	1X	1X	1X	1X	1X	1X
add	LI	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**
	BLI	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**
	RBLI	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**
	BLISS	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**
	BLISSDB	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**	**00:00**
	Speedup	1X	1X	1X	1X	1X	1X	1X	1X	1X	1X	1X
remove	LI	**00:02**	**00:03**	**00:04**	**00:05**	**00:06**	**00:07**	**00:09**	**00:11**	**00:13**	**00:16**	**00:18**
	BLI	**00:02**	**00:03**	**00:04**	**00:05**	**00:06**	**00:07**	**00:09**	**00:11**	**00:13**	**00:16**	**00:18**
	RBLI	**00:02**	**00:03**	**00:04**	**00:05**	**00:06**	**00:07**	**00:09**	**00:11**	**00:13**	**00:16**	**00:18**
	BLISS	**00:02**	**00:03**	**00:04**	**00:05**	**00:06**	**00:07**	**00:09**	**00:11**	00:14	**00:16**	00:19
	BLISSDB	**00:02**	**00:03**	**00:04**	**00:05**	**00:06**	**00:07**	**00:09**	**00:11**	**00:13**	**00:16**	00:19
	Speedup	1X	1X	1X	1X	1X	1X	1X	1X	1X	1X	1X

TABLE 8
Methods, Best Technique and Maximum Speedup

Method	Best technique	Speedup
BinHeap.bfs	BLISS/BLISSDB	36,000X
AvlTree.bfs	BLISSDB	326X
TreeSet.bfs	BLISSDB	188X
AvlTree.contains	BLI/RBLI/BLISS/BLISSDB	188X
TreeSet.repOK	BLISSDB	77X
AvlTree.dfs	BLISS/BLISSDB	50X
TreeSet.add	BLISSDB	43X
BinHeap.insert	BLISSDB	33X
BinHeap.repOK	RBLI	28X
BinHeap.dfs	RBLI	24X
TreeSet.contains	BLI/RBLI/BLISS/BLISSDB	14X
BinHeap.extractMin(Bug)	BLISS	12X
AvlTree.insert	BLISS	10X
BinHeap.trace_L2	BLISSDB	7X
AvlTree.remove	BLISS	6X
TreeMap.trace_L2	RBLI	5X
TreeMap.trace_L1	BLI/RBLI/BLISS/BLISSDB	5X
AvlTree.repOK	BLISSDB	5X
BinHeap.trace_L3	RBLI/BLISS	5X
TreeSet.dfs	RBLI	4X
BinHeap.trace_L1	BLISS	4X
BinHeap.delete	RBLI/BLISS/BLISSDB	4X
TreeMap.trace_L3	RBLI	3X
BinHeap.extractMin	RBLI /BLISS/BLISSDB	3X
BinTree.bfs	RBLI	3X
TreeSet.remove	BLI/RBLI	2X
BinTree.dfs	RBLI	2X
BinTree.count	BLI/RBLI	2X
BinTree.repOK	RBLI	1X
LinkedList.repOK	LI/BLI/RBLI/BLISS/BLISSDB	1X
LinkedList.add	LI/BLI/RBLI/BLISS/BLISSDB	1X
LinkedList.remove	LI/BLI/RBLI/BLISS/BLISSDB	1X

methods in this paper (each one of the latter using the bounds during BLI, RBLI, BLISS and BLISSDB analyses). Thus, the total bound computation time in this case (168:23), contributes 5:26 to each of the 31 performed analyses. Again, adding this time to our analysis times has a minor impact on most experiments.

All new experiments reported in this paper were run on an Intel Core i7-2600 processor with a 3.40 GHz clock speed and 8 GB DDR3 RAM, running Linux 3.2.0. All times are wallclock times as provided by SPF. 4 GB of heap memory were allocated for the Java virtual machine.

5.2 Experimental Results

Tables 1, 2, 3, 4, 5, 6, and 7 report the analysis times for techniques LI, BLI, RBLI, BLISS and BLISSDB, on all the aforementioned classes and methods, for various scopes. The lowest analysis times are highlighted, and the corresponding speedup is then reported as the quotient of the analysis time required by LI and the best analysis time among those reported for BLI, RBLI, BLISS and BLISSDB. Note that in some cases, LI runs out of memory while the other techniques do not. In those cases we report an infinite speedup (∞). This happens in 20 methods. Particularly interesting are the cases for methods TreeSet.bfs and AvlTree.bfs, where LI runs out of memory by scope 14 and 13, respectively, whereas BLISS and BLISSDB are able to reach scopes 17 and 20, respectively.

Looking at the speedups for method BinomialHeap.bfs in Table 4, we see that for scopes 15 through 20, ">" symbols pile up. For scope 15, the explanation for such notation is simple: if the actual analysis time was 10 hours, the speedup would be 36,000X. However, since the analysis timed out at 10 hours (but could have taken possibly much longer to complete), all that we can conservatively affirm is that the speedup is *at least* 36,000X. Analysis times typically grow exponentially as the scope is increased. For scope 16, although we can only guarantee a 36,000X speedup, due to the exponential growth in analysis times it is likely that the actual speedup is *much* larger (noted by $\gg$) than 36,000X. The same principle applies to even larger scopes; in order to remind the reader of the exponential growth in analysis times, we add another ">" symbol for each scope.

In Table 8 we sort the 32 methods in our experimental evaluation by maximum speedup achieved. The listing also includes the technique that yielded said speedup. We focus on those analyses where memory was not exhausted. Therefore, infinite speedups are dismissed. From Table 8 we see that 19 out of the 32 methods (59 percent) achieve a speedup greater than or equal to 5X. According to Table 10, for most of the methods where the speedup was below 5X, the reduction in the number of partially symbolic structures collected is significant. For example, as shown in Table 9, out of the 13 methods whose analysis speedup is below 5X, 6 reduce the corresponding set of partially symbolic structures by more than 50 percent. In fact, out of the 32 methods under analysis, 47 percent get their collected structures reduced by more than 50 percent. This will yield considerable savings in testing, since these structures will need to be solved to build test suites.

The remaining seven methods, in which low speedups and reductions on collected structures occurred, are the following:

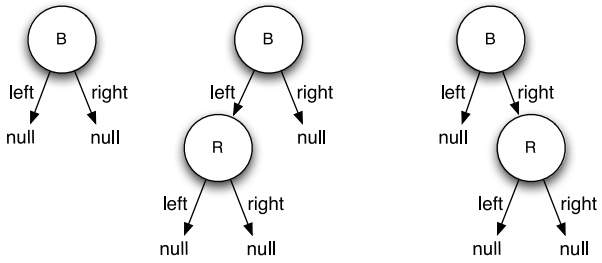
1) BinTree.bfs,
2) BinTree.dfs,
3) BinTree.repOK,
4) BinTree.count,
5) LinkedList.repOK,
6) LinkedList.add, and
7) LinkedList.remove.

The fact that class BinTree does not lend itself well to the techniques introduced in this paper should not be surprising. This is due to the fact that BinTree is a class whose adapted hybrid class invariant (characterizing whether a partially symbolic structure can be extended to a binary tree) can precisely determine the infeasibility of partially symbolic structures. Therefore, as discussed just before Section 5.1, there is no room for additional pruning benefits beyond those achieved by using the hybrid invariant within LI.

As for the methods from class LinkedList, their analysis times are almost negligible; therefore, there is hardly any room for optimization. Still, LinkedList is the only class in our benchmark that contains cyclic structures, and it serves the purpose of showing that the techniques can be applied to cyclic structures without problems or noticeable overhead.

```
public static void main(String[] args) throws Exception {
    BinomialHeap X = new BinomialHeap();
    X = (BinomialHeap) Debug.makeSymbolicRef("X", X);
    if (X != null && X.repOK_Concrete()){
        X.extractMin();
        if (X.size != X.numNodes()){
            throw new Exception();
        }
    }
}
```

Fig. 14. Method extractMin(bug find) used as driver for SPF.



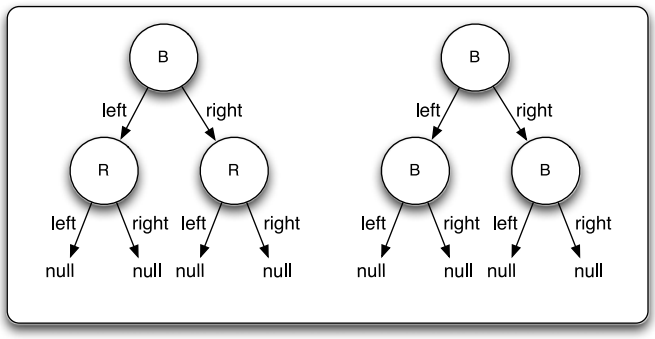


Fig. 15. Nonempty red-black trees with up to three nodes generated by Korat.

As shown in Fig. 10, we use the methods in the benchmark as drivers that lead the execution of SPF until the state space determined by the methods is exhausted. Notice that we are not looking for existing bugs; moreover, the try-catch block surrounding the method call masks any runtime exception that might be caught by SPF. Therefore, a valid question is to what extent the proposed techniques contribute toward finding bugs. Method extractMin in class BinomialHeap contains a nontrivial bug, first detected in [10]. Method extractMin(bug find) from class BinomialHeap denotes use of the "main" method shown in Fig. 14. This driver executes method extractMin until a state in which field `size` is incorrectly set is found. This bug requires a structure with at least 13 nodes to be exhibited. In Table 4, cells that correspond to experiments in which the bug is found are highlighted. In scope 13 we observe that our techniques produce a 2X speedup. It is interesting to note that when the scopes are increased, the speedup grows as well, reaching 12X for scope 20.

5.3 Implementation Details

BLISS and its related techniques were implemented on top of the standard distribution of SPF downloadable from [26]. The techniques presented in this paper are included as alternatives to LI, which is implemented in class GETFIELD. To guarantee that the LI experiments remain unbiased by the introduction of the new techniques, we left class GETFIELD untouched and introduced a new class GETFIELDBounded that incorporates the new techniques.

The standard SPF distribution [26] does not include a clear mechanism for introducing the hybrid preconditions. Executing the hybrid preconditions was necessary in order to make a fair evaluation. Therefore, we introduced a generic mechanism that allows the user to include hybrid preconditions in the class under analysis. These are executed during LI using Java's reflection mechanism.

5.4 Threats to Validity

BLISS requires the user to provide a representation invariant for the class under analysis in both imperative and declarative forms (e.g., as both a repOK method and a JML predicate). This requirement could be perceived as a limitation, especially in situations where one of the versions is available but the other one is not. Our experience, however, suggests that the hardest task is usually that of writing the first invariant in a correct and complete fashion, in either form. Once that is successfully accomplished, translating the correct and complete invariant to the other paradigm is a comparably much simpler matter.

While the techniques introduced in this paper were proven theoretically sound and complete, we have not verified the implementation as formally correct: the code may contain errors. However, we have checked that the experimental results are consistent across tools. In particular, the number of structures generated when analyzing method repOK is consistent with the number of structures generated by Korat [1] for all classes except TreeSet. The difference for class TreeSet is explained by the fact that all techniques in this paper (including LI) keep the node coloring symbolic. Therefore, trees with the same structure that differ only in coloring are collapsed into a single structure. For example, for scope 3 (i.e., structures with up to three nodes), Korat produces the five nonempty structures depicted in Fig. 15. If we instead look at the structures generated by SPF, we only get four nonempty structures. This is due to the fact that the two structures shown inside the box are collapsed by SPF into a single structure.

6 Related Work

Constraint-based bounded verification has its origins in [13], where a translation from annotated code to SAT is proposed and off-the-shelf SAT-solvers are used in order to determine the existence of bugs in the code under analysis. Several articles suggest improvements over [13]. For instance, [21] uses properties of functional relations to improve Java code analysis, and provides improvements for integer and array analyses. Bounded verification can be performed modularly, as shown in [8]. In [10], the use of tight field bounds allowed for a significant improvement on bounded verification, which we leveraged in [12], as well as in the techniques presented in this paper.

Symbolic execution and bounded verification were combined in [20]. Symbolic execution was used to build path conditions that were later solved using bounded verification. Bounds have also been used in the context of symbolic

TABLE 9
Number of Partially Symbolic Structures Collected by Each Technique, and Corresponding Reduction Compared to LI

Method	Technique	S03	S04	S05	S06	S07	S08	S09	S10	S11	S12
TreeSet.dfs	LI	8	22	64	196	625	2,055	6,917	23,713	82,499	OOM
	BLI	4	14	20	92	385	1,511	3,909	16,353	64,835	-
	RBLI	4	8	14	42	151	555	1,657	6,083	22,953	-
	BLISS(DB)	4	8	14	26	55	95	141	217	407	-
	Reduction	50%	63%	78%	86%	91%	95%	98%	99.1%	99.6%	
BinHeap.trace_L1	LI	40	119	349	1,049	OOM					
	BLI	31	77	251	659	-					
	RBLI	9	20	77	363	-					
	BLISS(DB)	25	43	130	547	-					
	Reduction	37%	63%	62%	47%						
BinHeap.delete	LI	16	58	196	647	OOM					
	BLI	12	37	144	416	-					
	RBLI	9	20	77	363	-					
	BLISS(DB)	9	18	68	327	-					
	Reduction	43%	68%	64%	49%						
TreeMap.trace_L3	LI	775	OOM								
	BLI/RBLI/BLISS(DB)	219	-								
	Reduction	71%									
BinHeap.extractMin	LI	8	19	45	117	291	OOM				
	BLI	6	14	38	98	254	-				
	RBLI/BLISS(DB)	5	8	20	96	252	-				
	Reduction	36%	57%	55%	17%	14%					
TreeSet.remove	LI	24	104	417	1,542	5,367	17,957	58,542	187,710	595,651	OOM
	BLI	8	56	100	643	2,988	11,912	27,395	106,510	388,824	-
	RBLI	8	47	91	535	2,414	9,642	22,538	87,156	317,473	-
	BLISS(DB)	8	47	91	493	2,229	8,933	20,242	79,621	OOM	-
	Reduction	66%	54%	78%	68%	58%	50%	65%	57%	46%	

execution; tools like Kiasan [6] and SPF [17] limit the length of reference chains. In [23] symbolic execution was used to generate tests for container classes closely resembling the ones used in this paper to assess our techniques. Several different approaches were used (in [23]) for test generation, including symbolic execution of repOK, but no relational field bounds were considered. All techniques that resort to symbolic execution can benefit from using the mechanisms associated with the techniques presented in this paper. For example the "lazier" [6] and "lazier#" [7] algorithms delay the concretization of a reference (much more so than standard lazy initialization), but eventually, when required, the approaches presented here can limit the number of choices for concretization.

Although symbolic execution is a white-box technique, it is worth mentioning that, when analyzing code that manipulates complex data, one can keep the structures concrete by taking a black-box approach to calling just methods that use the structures. Here, however, we are interested in doing symbolic execution of methods that take symbolic structures as input. For a detailed comparison of white-box versus black-box approaches to analyzing structures, the interested reader is referred to [23].

Green [24] is a technique that aims at providing a simple, canonical interface to a constraint solver in order to enable the recycling of results from one analysis run in future analysis runs. Although it is designed to be used in the context of symbolic execution, it targets the solving of path conditions. The DB component of BLISSDB cannot be easily substituted with Green due to the fact that the auxiliary solver checks used by BLISS (as explained in Section 4) are based on the translation of declarative invariants, not on path conditions.

7 Conclusions and Further Work

Relational field bounds have been successfully used in the context of bounded exhaustive bug finding, in order to increase analysis scalability. They have also been used for the improvement of generalized symbolic execution (symbolic execution extended to deal with programs that manipulate heap-allocated data structures) through an enhancement of lazy initialization called bounded lazy initialization [12]. In this paper, we built upon BLI and introduced novel techniques that further improve the efficiency of symbolic execution via two mechanisms: bound refinement and auxiliary feasibility (SAT) checks along the symbolic execution process. We showed that these mechanisms, jointly realized in a prototype called BLISS, significantly improve symbolic execution when compared to traditional LI and to BLI. Furthermore, we showed that BLISS can be improved even further by caching SAT checks, since many of these are repeated when carrying out the same analysis for different (typically increasing) scopes. We carried out

TABLE 10
Reductions in Test Suite Size Achieved by BLISS

Class	Method	MaxScope	#LI	#BLISS	Red. (%)
TreeSet	bfs	13	1,033,411	1,767	99.8
	dfs	11	82,499	407	99.5
	repOK	9	212	212	0
	contains	16	65,535	8,191	87.5
	add	9	141	141	0
	remove	10	187,710	79,621	57.5
TreeMap	trace_L1	10	209,343	77,662	62.9
	trace_L2	6	35,701	10,038	71.8
	trace_L3	3	775	219	71.7
AvlTree	bfs	12	290,511	425	99.8
	dfs	11	82,499	241	99.7
	repOK	12	425	425	0
	contains	15	32,767	511	98.4
	insert	9	1,640	477	70.9
	remove	7	2,297	396	82.7
BinHeap	bfs	14	4,240	15	99.6
	dfs	17	6,764	18	99.7
	repOK	20	21	21	0
	insert	8	224	184	17.9
	delete	6	647	327	49.5
	extractMin	7	291	252	13.5
	$\text{extractMin}_{\text{bug}}$	20	9	9	0
	trace_L1	6	1,049	547	47.9
	trace_L2	4	960	264	72.5
	trace_L3	2	103	84	18.5
BinTree	bfs	13	1,033,411	1,033,411	0
	dfs	13	1,033,411	1,033,411	0
	repOK	13	1,033,411	1,033,411	0
	count	13	1,033,411	1,033,411	0
LList	repOK	20	20	20	0
	add	20	2	2	0
	remove	20	16,234	16,234	0

experiments with classic data structure implementations that show the benefits of incorporating our techniques into Symbolic PathFinder, enabling the tool to effectively work with data structures whose size exceed the tool's previous capabilities (in time and/or space), with the goal of both systematically exploring program paths and automatically generating test inputs. Our experiments showed that, compared to LI and BLI, BLISS can reduce the time required to systematically explore program paths by up to four orders of magnitude, and that it generally reduces the number of structures obtained during path exploration (which have to be concretized using SMT solving to build test suites) by over 50 percent, with reductions of over 90 percent in some cases, compared to LI. We also showed that these reduced collections of partially symbolic structures retain exactly the same coverage as the much larger collections that would be obtained using LI, since our techniques only remove spurious structures.

As explained in Section 2.1.1, some constraints are harder to generalize (in order to admit partially symbolic structures) as hybrid preconditions than others. Even if some of the harder ones could perhaps be generalized by hand (albeit possibly at the cost of introducing new errors), we showed that some constraints do not lend themselves to be captured by a hybrid invariant at all. This can become a nontrivial obstacle for the usability of symbolic execution on programs dealing with heap-allocated structures, considering that a hybrid invariant is a necessary prerequisite of all the techniques involved, starting with (and including) traditional LI.

In this context, we conclude that the BLISS techniques are particularly effective for the verification of classes whose concrete repOK cannot be easily and/or completely captured by the hybrid invariant. Our experimental results show that BLISS obtains better results on classes with less precise hybrid invariants, where there is room for improvement, i.e., some distance between the concrete and hybrid invariants' pruning power that can be compensated by BLISS.

The new techniques require precomputed field bounds for the fields of the program under analysis. Computing tight field bounds as explained in [10] requires a large number of satisfiability queries, which are independent and can therefore be parallelized. Hence, a cluster is used to compute these bounds. We are working on alternative, more efficient ways of computing bounds. In particular, we are currently developing bound computation mechanisms that can be run on a single workstation, with efficiency comparable to the approach in [10], but which may lead to less precise (yet sound) bounds.

We also plan to integrate Green [24] into the BLISS distribution. As explained in Section 6, Green is not a practical substitute for the non-path-condition-related SAT checks used by BLISS (which are cached by the BLISSDB mechanism). Nevertheless, it could be a useful addition towards obtaining the benefits of verdict caching on the SMT side (i.e., to recycle path-condition-related SMT check results across runs) as well. In particular, there are some cases where the benefits of BLISS are eclipsed by a proportionally large amount of runtime being invested in the SMT-solving of path conditions. Incorporating Green could be an important step towards improving effectiveness in such cases.

The techniques we presented aim at producing a complete exploration of the state space. Yet, for instance, in the context of bug finding, it could be more effective to replace such completeness with new techniques to explore larger structures. It might perhaps be useful to use overly refined TACO bounds from larger scopes, which may allow us to explore new structures at the expense of pruning valid instances.

Acknowledgments

The authors thank Elena Morin for proofreading this paper. This publication was made possible by NPRP grant number NPRP-4-1109-1-174 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors. Marcelo F. Frias is the corresponding author.

References

[1] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2002, pp. 123–133.

[2] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, pp. 209–224, 2008.

[3] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmanni, and W. Visserr, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 1066–1071.
[4] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," in *Proc. 4th Int. Conf. Formal Methods Components Objects*, 2005, pp. 342-363.
[5] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
[6] X. Deng, J. Lee, and Robby, "Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2006, pp. 157–166.
[7] X. Deng, Robby, and J. Hatcliff, "Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs," in *Proc. 5th Int. Conf. Softw. Eng. Formal Methods*, 2007, pp. 273–282.
[8] G. Dennis, K. Yessenov, and D. Jackson, "Bounded verification of voting software," in *Proc. 2nd Int. Conf. Verified Softw.: Theories, Tools, Exp.*, 2008, pp. 130–145.
[9] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata, "Extended static checking for Java," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2002.
[10] J. P. Galeotti, N. Rosner, C. López Pombo, and M. F. Frias, "Analysis of invariants for efficient bounded verification," in *Proc. 19th Int. Symp. Softw. Testing Anal.*, 2010, pp. 234–245.
[11] J. P. Galeotti, N. Rosner, C. López Pombo, and M. F. Frias, "TACO: Efficient SAT-based bounded verification using symmetry breaking and tight bounds," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1283–1307, Sep. 2013.
[12] J. Geldenhuys, N. Aguirre, M. F. Frias, and W. Visser, "Bounded lazy initialization," in *Proc. 5th Int. NASA Formal Methods Symp.*, 2013, pp. 229–243.
[13] D. Jackson, and M. Vaziri, "Finding bugs with a constraint solver," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2000, pp. 14–25.
[14] S. Khurshid, C. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proc. 9th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2003, pp. 553–568.
[15] J. C. King "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[16] M. Staats and C. S. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proc. 19th Int. Symp. Softw. Testing Anal.*, 2010, pp. 183–194
[17] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 391–425, 2013.
[18] Redis can be downloaded from. (2015). [Online]. Available: http://redis.io/download
[19] N. Rosner, J. P. Galeotti, S. Bermúdez, G. M. Blas, S. Perez De Rosso, L. Pizzagalli, L. Zemín, and M. F. Frias, "Parallel bounded analysis in code with rich invariants by refinement of field bounds," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 23–33.
[20] D. Shao, S. Khurshid, and D. Perry, "Whispec: White-box testing of libraries using declarative specifications," in *Proc. Symp. Library-Centric Softw. Des.*, 2007, pp. 11–20.
[21] M. Vaziri and D. Jackson, "Checking properties of heap-manipulating procedures with a constraint solver," in *Proc. 9th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2003, pp. 505–520.
[22] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.
[23] W. Visser, C. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2004, pp. 97–107.
[24] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, reusing and recycling constraints in program analysis," in *Proc. ACM SIGSOFT Symp. Foundations Softw. Eng.*, 2012, pp. 58–69.
[25] (2015). [Online]. Available: http://www.satcompetition.org/2009/format-benchmarks2009.html
[26] (2015). [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc

Nicolás Rosner is currently working toward the doctoral degree at the Department of Computer Science, School of Science (FCEyN), Universidad de Buenos Aires, Argentina, where he is a lead teaching assistant. His research interests include software verification and test suite generation based on constraint solving as well as distributed systems.

Jaco Geldenhuys received the BSc and MSc degrees in computer science from Stellenbosch University, and the DTech degree in computer science from the Tampere University of Technology. He is currently an associate professor of computer science at Stellenbosch University. His research focuses on software engineering, specifically formal methods (model checking and process algebra), static analysis, testing, and open source software.

Nazareno M. Aguirre received the PhD degree from King's College London, University of London, United Kingdom. He is currently an associate professor at the Computer Science Department, University of Rio Cuarto, Argentina, and a researcher of the Argentinian National Council for Scientific Research (CONICET). His research interests include the foundations of software engineering, and formal techniques applied to software analysis.

Willem Visser is currently a professor of computer science at Stellenbosch University, South Africa. Before joining Stellenbosch in 2009, he spent eight years at NASA Ames Research Center, where he was one of the research leads for the Java PathFinder project. His research interests include model checking, testing, symbolic execution, and model counting. He has been the co-chair of ASE in 2008 and will be ICSE program co-chair in 2016. He is also currently on the steering committee for ICSE and SPIN, on the executive committee of ACM SIGSOFT and is a member of the editorial board of *ACM Transactions on Software Engineering and Methodology*.

Marcelo F. Frias is currently a professor of computer science at the Buenos Aires Institute of Technology and a researcher at CONICET. His research interests include 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.