Bounded Lazy Initialization

Jaco Geldenhuys[1], Nazareno Aguirre[2,4], Marcelo F. Frias[3,4] and Willem Visser[1]

[1] Computer Science Division, Department of Mathematical Sciences, Stellenbosch University, Private Bag X1, 7602 Matieland, South Africa.
E-mails: {jaco, wvisser}@cs.sun.ac.za.
[2] Department of Computer Science, FCEFQyN, Universidad Nacional de Río Cuarto (UNRC), Argentina. E-mail: naguirre@dc.exa.unrc.edu.ar.
[3] Department of Computer Engineering, Instituto Tecnológico de Buenos Aires (ITBA). E-mail: mfrias@itba.edu.ar.
[4] National Scientific and Technical Research Council (CONICET), Argentina.

Abstract. Tight field bounds have been successfully used in the context of bounded-exhaustive bug finding. They allow one to check the correctness of, or find bugs in, code manipulating data structures whose size made this kind of analyses previously infeasible. In this article we address the question of whether tight field bounds can also contribute to a significant speed-up for symbolic execution when using a system such as Symbolic Pathfinder. Specifically, we propose to change Symbolic Pathfinder's lazy initialization mechanism to take advantage of tight field bounds. While a straightforward approach that takes into account tight field bounds works well for small scopes, the lack of symmetry-breaking significantly affects its performance. We then introduce a new technique that generates only non-isomorphic structures and consequently is able to consider fewer structures and to execute faster than lazy initialization.

1 Introduction

Many techniques have been devised in order to determine to what extent a software artifact is correct. Testing [1], for instance, is one of these techniques. Two main reasons justify the place testing occupies in most software development projects: it is lightweight, and it is scalable. The downside, as is well-known, is that testing only allows one to detect errors that occur when code is executed on the tested inputs. In order to achieve greater guarantees of software correctness, more conclusive program analysis techniques have to be considered. For instance, bounded verification [7] and model checking [2] guarantee that no errors can be exhibited on significantly larger input sets (i.e., on that part of the input state space that was successfully explored by the corresponding technique), compared to testing. This is achieved, of course, at the expense of scalability. Therefore, improving the scalability of the latter analysis techniques is a *must*.

Bounded exhaustive verification automatically checks code correctness, but subject to a *scope*, consisting of a maximum number of iterations and object instances for the classes involved [7, 4, 5]. Therefore, if the technique is successful in verifying code, it does not guarantee absolute correctness, but correctness within

the established scope (i.e., no errors exist that require at most the established maximum number of iterations, and involve at most the established number of object instances). As shown in [5], by appropriately bounding the values that class fields can take, bounded exhaustive verification based on SAT-solving can be significantly improved. In particular, field bounds allowed bounded exhaustive verification to significantly increase data domains scopes for analysis, and to detect bugs that other tools based on bounded verification, model checking, or SMT-solving failed to detect [5]. Here, we explore whether by using field bounds one can also improve the scalability of symbolic execution of structures, as performed by Symbolic PathFinder (SPF) [10], an extension of Java PathFinder (JPF).

The field bounds considered in this work are the *tight bounds* computed by the approach in [5], which uses the structural invariants of the classes under analysis (the so-called `repOK`). Intuitively, by changing SPF's lazy initialization approach [8], where all possible aliasing possibilities are explored, to only take into account those included in pre-computed field bounds, considerably fewer structures should be considered. Interestingly, as the results for binary search trees in Section 5 show, this intuition holds for structures up to 6 nodes, but then lazy initialization starts to perform better. As it will be discussed later on, it turns out that the benefit of considering fewer options for each reference to be lazily initialized is outweighed by the fact that the bounded approach considers isomorphic structures whereas lazy initialization does not. That is, although lazy initialization constructs many structurally invalid structures, they are quickly pruned by `repOK`, whereas in the bounded case duplicate (isomorphic) valid structures are considered throughout the analysis and are never pruned.

The above problem is overcome by a new algorithm, introduced in this paper, that not only bounds lazy initialization, but also produces only non-isomorphic structures. This algorithm can be shown to strictly consider fewer structures than lazy initialization, since it behaves similarly except that some aliasing options are not considered. This algorithm constitutes the main contribution of the paper.

Contributions. In this paper we make the following contributions:

1. We study of the usefulness of field bounds in the context of symbolic execution of structures.
2. We show that symmetry-breaking, as a mechanism to prevent considering isomorphic structures, is important for efficiency.
3. We propose an algorithm that incorporates field bounds with symmetry breaking into symbolic execution, and implement it within SPF.
4. We assess the above on three classic data structures: linked lists, binary trees and red-black trees.

Structure of the article. In Sections 2 and 3, we introduce and discuss field bounds and lazy initialization, respectively. In Section 4 we present the notion of *Bounded Lazy Initialization*, and introduce both a straightforward algorithm for it, and a more efficient one that performs symmetry-breaking. In Section 5, we

present experimental results showing that, on a relevant data structures benchmark, bounded lazy initialization with symmetry breaking scales better than standard lazy initialization. In Section 6 we discuss related work, and in Section 7 we present our conclusions and proposals for future work.

2 Tight Field Bounds and Program Analysis

Bounded program verification was introduced in [7] as a technique for bug detection. In [4, 5] it is implemented by translating Java code annotated with contracts to a propositional formula, which is solved using a SAT-solver. This approach requires the engineer to provide a *scope*, consisting of a maximum number of iterations and object instances for the classes involved [7, 4, 5]. The existence of a satisfying valuation for the formula can then be traced back to an execution exhibiting an erroneous behavior (unhandled exception or contract violation) within the provided scope. If no valuation is found, we know that the method is *correct within the prescribed scope*.

The encoding of bounded program correctness as a satisfiability problem involves interpreting programs in terms of relations. Given a class `C`, a class field `f` of type `C'` defined in `C` can be semantically interpreted, in a given program state, as a total function f mapping object references from C (the semantic domain associated with class `C`), to C' (the domain associated with class `C'`). That is, in a given program state, f can be seen as a binary relation contained in $C \times C'$. Notice that properties of the state may make some tuples of $C \times C'$ infeasible as part of the interpretation of a field `f`. In particular, if the state is assumed to satisfy certain representation invariant (e.g., the states prior to the execution of the code under analysis are assumed to satisfy a precondition, which would include the wellformedness of the inputs), all tuples corresponding to ill-formed structures will necessarily be out of the semantic interpretation of `f` in that state. For instance, in linked lists, if the representation invariant indicates that lists must be acyclic, then tuples of the form $\langle N, N \rangle$, with $N \in \mathit{Node}$ (the semantic domain associated with the `Node` class) cannot belong to *next* (the semantic interpretation of `next`), if the corresponding state is assumed to satisfy the invariant.

The above observation about infeasible tuples in fields' semantic domains can be enhanced if one is able to prevent isomorphic structures via symmetry-breaking. Symmetry breaking enforces a canonical ordering on the way references are stored in the model of the memory heap used during analysis. In [5], a mechanism for automatically defining symmetry-breaking predicates, for any Java class, is introduced. These predicates force assigning node references to structures following a breadth-first ordering. Figure 1 shows examples of the ordering for singly linked lists and binary trees. The ordering imposed by symmetry-breaking predicates prevents some references from being connected. For instance, in a list, a node reference N_i can only point (through field `next`) to N_{i+1} or the value `null`, if symmetry-breaking is imposed. Similarly, reference N_0 can only point through field `left` to N_1 or the value `null` (any other candidate would

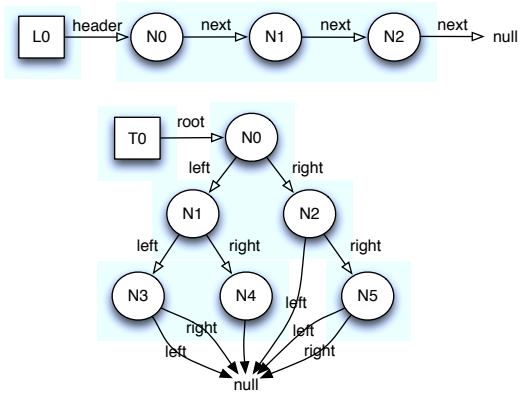

Fig. 1. Node ordering for list-like and tree-like data structures.

violate the breadth-first ordering). If we instead consider field `right`, N_0 can point to N_1 (only in case N_0.`left` = `null`), N_2, or `null`.

Also in [5], the notion of *tight field bound* is introduced. Tight field bounds allow us to remove from fields' semantic domains the tuples that are infeasible due to representation invariants and symmetry-breaking, and lead to exponential speed-ups in analysis. Let us describe this notion. Let us consider a class `C` and a field `f` in `C`, of type `C'` (i.e., declared as `C' f`). A *scope* determines sets C and C' of object instances of classes `C` and `C'`, respectively (e.g., if the scope for class `Node` is 3, then $Node = \{N_0, N_1, N_2\}$). Notice that the scope does not determine the set of objects live at a specific runtime configuration, but rather the runtime objects in any configuration. A sample scope would be, for instance, 7 `Node` (e.g., for performing a bounded verification on all linked lists, or binary search trees, composed of up to 7 nodes). Consider the lattice of binary relations $\langle \mathcal{P}\left(C \times (C' \cup \{\texttt{null}\})\right), \subseteq \rangle$ (disregard `null` if `C'` is a basic type). A *tight bound* for field `f` is a member $U_{\texttt{f}}$ of $\mathcal{P}\left(C \times (C' \cup \{\texttt{null}\})\right)$ in which all pairs must belong to some semantic interpretation for field `f`. Since the semantic interpretations might be constrained by certain state properties (e.g., representation invariants or symmetry breaking predicates), some tuples might necessarily be out of tight field bounds.

As an example, consider scope `4 Node` for acyclic linked lists with symmetry-breaking. Then, relation

$$\{(N_0, N_1), (N_0, \texttt{null}), (N_1, N_2), (N_1, \texttt{null}), (N_2, N_3), (N_2, \texttt{null}), (N_3, \texttt{null})\}$$

is a tight bound for field `next`. Similarly, relation

$$\{(N_0, N_1), (N_0, \texttt{null}), (N_1, N_2), (N_1, N_3), (N_1, \texttt{null}), (N_2, N_3), (N_2, \texttt{null}), (N_3, \texttt{null})\}$$

is a tight bound for field `left` of binary search trees, under scope `4 Node`. We invite the reader to refer to Figure 1 to verify that pair (N_0, N_2), for instance,

cannot belong to the tight bounds for fields `next` (resp., `left`) of linked lists (resp., binary search trees) under symmetry breaking.

To take advantage of tight field bounds, these must be computed prior to actual program analysis. In [5], an effective distributed algorithm for computing tight field bounds is presented. The implementation, that is designed using a master/slave architecture, allows for the removal, from the field's semantic domain, of those pairs that cannot belong to any valid instance that satisfies the symmetry breaking-induced ordering and other constraints such as a representation invariant. Precomputing field bounds contributes to the scalability of analysis, since bounds only depend on the class, its invariant and the scope, but are independent from the code of the method under analysis. Also, once the bounds are computed, they are stored in a bounds database, and often reused. For instance, the same bound can be used for the analysis of all the methods in a class, and for different kinds of analysis (verification, test input generation, etc.). Therefore, the cost of computing bounds is amortized by their frequent use.

3 Lazy Initialization

Lazy initialization [8] is a technique for symbolic execution especially tailored for handling complex, possibly unbounded data structures. Symbolic execution begins with uninitialized field values and, along symbolic execution of a method `m`, class fields are initialized only when they are accessed. Whenever a (previously uninitialized) field `f` for an existing object o is accessed, the lazy initialization for o.`f` takes place. When o.`f` is a reference to an instance of object type, the following possibilities are, non-deterministically, considered:

- o.`f` may take the value `null`,
- o.`f` may refer to an already existing object,
- o.`f` refers to a new object.

This is formalized in the algorithm presented in Figure 2, extracted from [8]. In order to make the contribution of this paper more clear we separate the above choices into a function called *options()* which will be adapted in the following sections.

3.1 A Running Example

Let us consider the algorithm in Figure 3. This algorithm searches for an integer value stored in a `TreeSet`. It returns the node that stores the valuer, if the value is stored in the `TreeSet`, or it returns `null` otherwise. Figure 4 portrays 13 out of the 57 structures generated by the Lazy Initialization mechanism along the symbolic execution of method `Contains` (we present those structures where only fields `root` or `left` are being initialized). In Section 4, where we introduce *Bounded Lazy Initialization*, we will come back to this example in order to compare the number of generated structures.

```
if (f is uninitialized) {
  if (f is a reference field of type T) {
    nondeterministically initialize f to each element of options()
    if (method precondition is violated) {
      backtrack()
    }
  }
  if (f is primitive (or string) field) {
    initialize f to a new symbolic value of appropriate type
  }
}


function options()
  return the set consisting of
    1. null
    2. a new object of class T (with uninitialized field values)
    3. every object created during a prior initialization of a field of type T
```

Fig. 2. Lazy Initialization Algorithm.

4 Bounded Lazy Initialization

Bounded Lazy Initialization profits from the existence of pre-computed tight field bounds in order to prune the state space exploration performed by lazy initialization even further. Let us consider an object o and a field `f` such that o.`f` is next to be lazily initialized. For the sake of intuition, consider the partially initialized `TreeSet` from Figure 5, with $o = N_1$ and `f` = `left`. According to Figure 2, the following possibilities arise during lazy initialization:

- o.`f` = `null`,
- o.`f` = N_2, with N_2 a new uninitialized node, or
- o.`f` may refer to N_0 or N_1.

The tight bound for `TreeSet` field `left` with up to 3 nodes is

$$\{(N_0, \texttt{null}), (N_0, N_1), (N_1, \texttt{null}), (N_2, \texttt{null})\} .$$

Out of the 4 alternatives that would be explored using lazy initialization, only one is feasible according to the bound, namely, initializing o.`f` = `null`. The remaining options introduce tuples to field `left` that were already deemed infeasible by the bound pre-computation. Certainly, initializing o.`f` = N_0 or o.`f` = N_1 leads to a cyclic structure, and therefore these initializations are correctly prevented by the bounds. Even more interesting, o.`f` = N_2 is also prevented since the resulting structure would become unbalanced, with no nodes remaining to regain the balance. It is worth noticing that tight field bounds capture these subtleties that elude lazy initialization.

```
public TreeSetNode Contains(int key) {
        TreeSetNode p = root;
        while (p != null) {
                if (key == p.key) {
                        return p;
                }
                else if (key < p.key) {
                        p = p.left;
                }
                else {
                        p = p.right;
                }
        }
        return null;
}
```

Fig. 3. Method `Contains` from class `TreeSet`.

Unlike lazy initialization, bounded lazy initialization bounds the size of the generated structures. Lazy initialization produces partially initialized structures that eventually have to be made concrete. The concretization process may require generating a structure that, because of its size, exceeds the capabilities of the concretization technique. Therefore, whenever possible, keeping the size of the structures under control is beneficial for analysis.

Bounded Lazy Initialization modifies the lazy initialization algorithm by filtering those initializations that are incompatible with the tight field bounds. In Section 4.1 we present a first approach to Bounded Lazy Initialization that is particularly useful for explaining the concept, as well as for exposing a limitation that is later on addressed, in Section 4.2.

4.1 First Approach: Initializing from Bounds

The first algorithm for Bounded Lazy Initialization is given in Figure 6. When a field `f` has to be initialized, the algorithm allows one to consider all the options provided by the tight bound for field `f`. In Figure 7, we show all the structures produced by Bounded Lazy Initialization during the symbolic execution of method `Contains` (cf. Figure 3). There are clear differences with the outcome of lazy initialization (cf. Figure 4). The reduction on the number of generated structures (57 for lazy initialization versus 13 for its bounded version) is obviously significant. Besides, as argued above, this "pruning" is sound, since the structures that are no longer produced by the bounded lazy initialization procedure stand no chance of satisfying the class invariant.

An important property of lazy initialization is that no isomorphic partially initialized structures are ever generated. Therefore, no obviously redundant structures are being produced. When we move to bounded lazy initialization, this

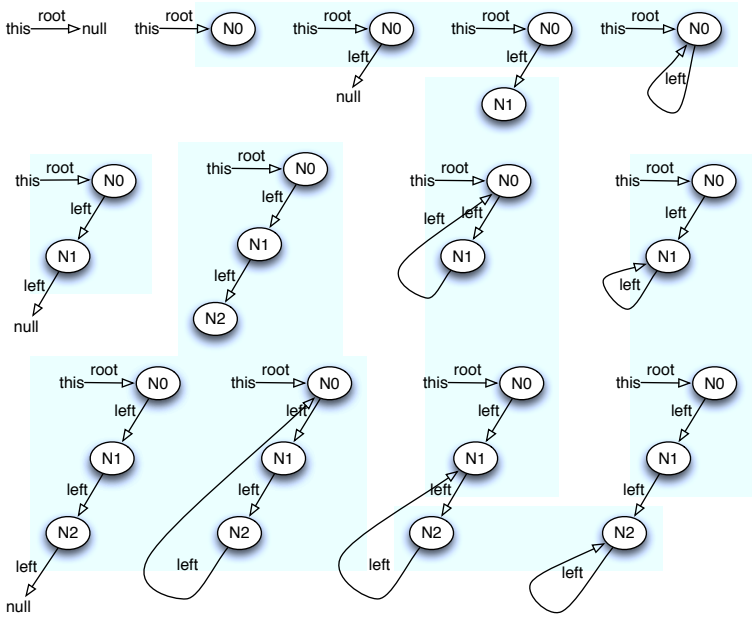

Fig. 4. Some of the structures generated by lazy initialization along the symbolic execution of method `Contains`.

property is lost. Notice in particular that in Figure 7, the 6th and the 7th partially initialized structures (also shown on the left side of Figure 8) are indeed isomorphic (also the 10th and 12th as well as 11th and 13th). Unfortunately, the number of isomorphic structures grows to a point where the advantage of using the bounds is seriously reduced. In Section 4.2 we present an alternative approach that follows the same intuition, yet avoids producing isomorphic partially initialized structures.

4.2 Second Approach: Regaining Full Symmetry Breaking

Let us analyze the 6th and 7th structures from Figure 7 (see left part of Figure 8). The reason for having these two isomorphic initialization alternatives for N_0.`right` is that by making use of the information provided by the tight bound for field `right`, the options for this field, for node N_0, are `null`, N_1, or N_2. However, it is not necessary to consider two different "non-null" initializations. In order to avoid these isomorphic structures, we will use sets of references as labels for nodes in the partially initialized structure. Figure 8 illustrates how the structures get merged into a common structure under this new approach. The intuition is the following: *each node is labeled with a set of references that can be reached by traversing the fields, and are compatible with the tight field bounds.* The new algorithm for bounded lazy initialization is presented in Figure 9. Let

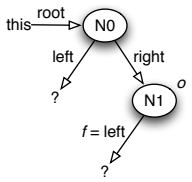

Fig. 5. A partially initialized `TreeSet` instance.

Input: Receiver object this, and field `f`
Input: Tight bound $U_{\mathtt{f}}$ for field `f`
  function options()
    return $\{t : \text{T such that } (\text{this}, t) \in U_{\mathtt{f}}\}$

Fig. 6. The Bounded Lazy Initialization algorithm (version 1).

us consider a node n in the partially initialized structure whose label set is N. Let `f` be the field that has to be initialized, and let $U_{\mathtt{f}}$ be its tight bound. Since $U_{\mathtt{f}}$ is a binary relation, we can compute $N' = N; U_{\mathtt{f}}$ (N' is then the set of all images of elements in N, with respect to relation $U_{\mathtt{f}}$). As it was the case for lazy initialization, the new algorithm also considers three cases, whose discussion follows:

- If $\mathtt{null} \in N'$, there must be a reference $r \in N$ such that $\langle r, \mathtt{null}\rangle \in U_{\mathtt{f}}$. Therefore, there may be a concrete structure instance in which $n.\mathtt{f} = \mathtt{null}$. Thus, `null` is a candidate definition that has to considered. Equally important, if $\mathtt{null} \notin N'$ there cannot be any node pointing to `null`. Therefore, `null` does not need to be considered. Notice that the lazy initialization algorithm *always* evaluates the possibility of using `null`.
- If N' contains some reference, then we consider adding a new node to the structure. Notice that if $N' = \{\mathtt{null}\}$, we do not add a new node. This decision, consistent with the tight bound information, prunes options that are unnecessarily considered by the lazy initialization algorithm.
- In lazy initialization, the third case initializes `f` as pointing to previously introduced nodes. But, if N' does not intersect the label set for a previously introduced node m, it is not possible (due to the bound induced constraints), that $\mathit{this}.\mathtt{f} = m$. This prunes initialization options that are currently considered by lazy initialization.

Notice that no isomorphic partially initialized structures can ever be generated. This immediately follows from the fact that we are generating structures in the same order lazy initialization does, yet we are skipping (probably many) initializations. If we use the algorithm from Figure 9, out of the 13 partially initialized structures considered in Figure 7 only 10 remain.

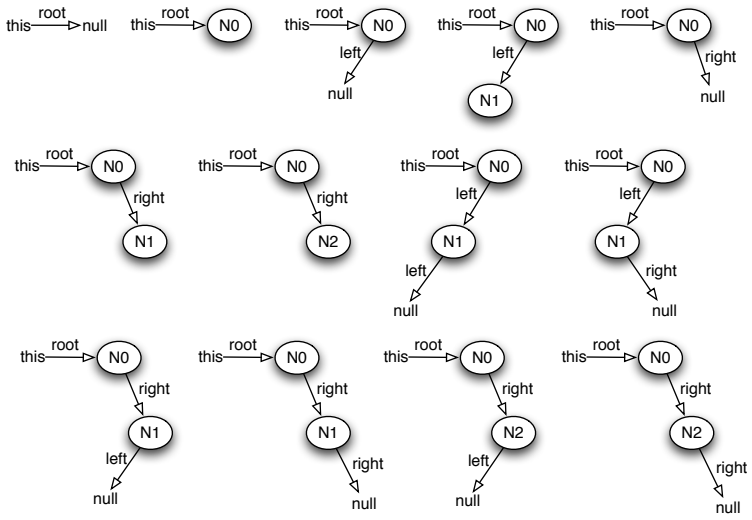

Fig. 7. The 13 structures generated by Bounded Lazy Initialization along the symbolic execution of method `Contains`.

5 Evaluation

The bounded lazy initialization algorithms described in the previous section were implemented in Symbolic PathFinder [10], and compared to the already-implemented lazy initialization algorithm. Three data structures are used to illustrate the performance of the new approach:

- **LList:** An implementation of sequences based on singly linked lists;
- **BSTree:** A binary search tree implementation from [14]; and
- **TreeSet:** An implementation based on red-black trees as found in `java.util`.

They cover linear and tree-like structures. Since the number of partially initialized structures generated during bounded lazy initialization strongly depends on the cardinality of the tight bounds, it is relevant to analyze the impact of the technique on heavily constrained structures (such as `TreeSet`, where tight bounds are smaller) and on less constrained structures (such as `BSTree`).

5.1 Experimental Setting

Tight field bounds were not computed as part of our experiments. Instead, precomputed databases for the data structures were reused. Computing tight field bounds, as put forward in [5], requires checking, via SAT solving, the feasibility of each tuple in the corresponding field's semantic domain. Thus, a high number of SAT queries, which depends on the scope, must be performed. However, these checks are all independent from one another, and therefore are subject to

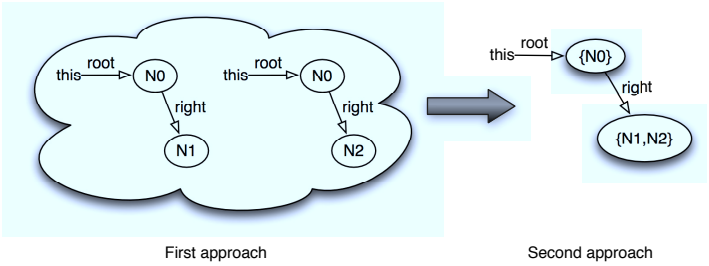

Fig. 8. From isomorphic partially initialized structures to a single partially initialized structure.

Input: Receiver object this, with node set N as label
Input: Tight bound $U_{\mathtt{f}}$ for field `f`
function options()
Let N' be the node set $N; U_{\mathtt{f}}$. **return** the set
1. `null`, if `null` belongs to N'
2. a new object of class T (with uninitialized field values), if $N' \setminus \{\mathtt{null}\} \neq \emptyset$
3. every object created during a prior initialization of a field of object type T whose label node set intersects with N'

Fig. 9. The Bounded Lazy Initialization algorithm (version 2).

parallelization. Indeed, in [5] the approach to compute tight field bounds uses a cluster. As a sample, the time required to compute tight bounds for lists, binary search trees and red black trees using the approach in [5] is 68:53, 00:38 and 02:51 (in minutes and seconds, mm:ss), for scopes 100, 12 and 12, respectively, and using a cluster of 16 quad-core PCs. These scopes exceed those used in this paper, and therefore the corresponding tight bound computation times serve as upper bounds of the actual times for the experiments in the paper. Each PC in the cluster had two Intel Dual Core Xeon 2.67 GHz processors, a 2 MB L2 cache, and 2 GB of RAM. The cluster used Debian GNU/Linux (kernel 2.6.18-6) and the Argonne National Laboratory's MPICH2 for message-passing.

The results reported in the rest of this section were computed on an Apple MacBook Pro with a 2.3 GHz Intel i5 processor with 4 Gb of memory, running the Mac OS X 10.8.2 operating system and the Darwin 12.2.0 kernel.

5.2 Experimental Results

Each experiment explores the execution of `repOK`(t) on a symbolic data structure t with n nodes. The value of n is a parameter for the experiments. The `repOK` routine checks that t satisfies the constraints on the wellformedness of the corresponding data structure; for example, in the implementation of `TreeSet`, `repOK` makes sure that t is a valid binary search tree, that node parent pointers are

		LI		BLI1			BLI2	
n	unique	explored	time	explored	duplicates	time	explored	time
1	1	2	<00:01	1	0	<00:01	1	<00:01
10	10	74	<00:01	19	0	<00:01	19	<00:01
100	100	5 249	00:02	199	0	<00:01	199	<00:01

Table 1. Experimental results for `LList`

		LI		BLI1			BLI2	
n	unique	explored	time	explored	duplicates	time	explored	time
1	1	4	<00:01	2	0	<00:01	2	<00:01
2	3	21	<00:01	10	0	<00:01	11	<00:01
3	8	82	<00:01	36	1	<00:01	44	<00:01
4	22	306	<00:01	145	9	<00:01	164	<00:01
5	64	1 140	00:01	668	61	00:01	639	00:01
6	196	4 275	00:02	3 554	393	00:02	2 464	00:01
7	625	16 144	00:04	21 165	2 523	00:05	9 604	00:03
8	2 055	61 332	00:09	140 996	16 927	00:17	35 695	00:06
9	6 917	234 154	00:29	1 030 989	119 747	01:43	136 260	00:16
10	23 713	897 596	01:44	8 259 479	908 563	13:47	516 376	00:53
11	82 499	3 452 526	06:34	–	–	–	1 972 260	03:12

Table 2. Experimental results for `BSTree`

correct, and that the red-black color constraints are satisfied, resulting in a balanced tree. As `repOK` traverses the data structure, the fields are initialized using the lazy, bounded lazy, and the symmetry-breaking bounded lazy techniques.

Tables 1, 2, and 3 (for `LList`, `BSTree`, and `TreeSet`, respectively) show the number of structures explored, and the execution times (in minutes and seconds, mm:ss). The last two tables show only those experiments that completed in less than 30 minutes. The results for `LList` in Table 1 do not convey much information. The first two columns show the value of n (the number of nodes) and the number of unique data structures of this size. As expected, these values are identical in the case of `LList`. The next three major columns show the results for the LI (lazy initialization), BLI1 (the first bounded lazy initialization), and BLI2 (the second bounded lazy initialization) techniques. The times, shown in columns 4, 7, and 9, are negligible. The values in columns 3, 5, and 8 are the number of choices made during the exploration according to the algorithms in Figures 2, 6, and 9, respectively. Because the last two values are bounded, not many such choices are explored; LI is entirely unconstrained and make all possible choices. Nevertheless, the times remain small.

The results for `BinTree` show a different case, since the number of choices is much larger, and the `repOK` implementation is more involved. Up to $n = 6$, LI makes more choices because it is not constrained by bounds. However, at $n = 7$ it is overtaken by BLI1 in this regard, because of the number of duplicates the latter explores. The number of duplicates explored (over and above the unique

		LI		BLI1			BLI2	
n	unique	explored	time	explored	duplicates	time	explored	time
1	2	4	00:01	2	0	<00:01	2	<00:01
2	4	27	<00:01	20	2	<00:01	15	<00:01
3	7	110	00:01	22	1	<00:01	21	<00:01
4	15	409	00:01	90	1	00:01	101	00:01
5	29	1 509	00:04	239	14	00:02	158	00:01
6	49	5 610	00:08	1 231	58	00:05	883	00:04
7	84	21 043	00:27	7 636	178	00:23	4 715	00:13
8	148	79 530	02:14	51 291	576	03:13	16 146	00:53
9	270	302 402	11:51	267 750	1 775	27:11	39 583	02:59
10	518	–	–	–	–	–	149 133	17:11

Table 3. Experimental results for `TreeSet`

structures) is given in the middle column of the table. This extra work is also reflected in the execution times. The BLI2 technique explores fewer choices than either LI and BLI1, meaning that it can analyze significantly more structures in the same amount of time. This same trend is also clear in the case of `TreeSet`.

6 Related Work

Constraint based bounded verification has its origins in [7], 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 [7]. For instance, [12] 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 [4]. In [5], the use of tight field bounds allowed us to improve bounded verification significantly.

Symbolic execution [9] is a technique for program analysis that executes a path in the program control flow graph using symbolic values. During the symbolic execution, conditions from branching statements are conjoined into a *path condition*. The satisfaction of the path condition allows one to create inputs that exercise the symbolically executed path. Lazy initialization [8] is an optimization of symbolic execution where dynamically allocated data structures are partially initialized on demand, deferring the initialization process as much as possible. Dynamic symbolic execution [6] (also called *concolic* execution), uses concrete executions to guide the symbolic execution phase.

Symbolic execution and bounded verification were combined in [11]. Symbolic execution was used to build path conditions that were later on solved using bounded verification. Bounds have also been used in the context of symbolic execution; tools like Kiasan [3] and Symbolic Pathfinder [10] bound the length of reference chains. In [13] symbolic execution was used to generate tests for containers similar to those used here. Various different approaches were used for

test generation, including symbolic execution of repOK(), but no bounds were considered. All the techniques that resort to symbolic execution may profit from using a mechanism such as, bounded lazy initialization, as defined in this paper.

7 Conclusions and Future Work

Tight field bounds have been successfully used in the context of bounded-exhaustive bug finding, in order to increase this analysis' scalability. In this paper, we studied whether field bounds can also contribute to improve the efficiency of symbolic execution. We showed not only that field bounds can be employed to improve the symbolic execution of structures, but also that symmetry breaking, as a mechanism to prevent considering isomorphic structures, is important for efficiency. We proposed two algorithms that incoporate field bounds into Symbolic Pathfinder's lazy initialization, resulting in what we call *bounded lazy initialization*. The first is a straightforward extension of lazy initialization to take into account field bounds, whereas the second prevents the generation of isomorphic structures. We carried out experiments with classic data structure implementations, that show the usefulness of our approach, and the importance of avoiding generating isomorphic structures.

The presented approach requires pre-computing tight bounds for the fields of the program under analysis. Computing tight field bounds, as put forward in [5], requires a high number of satisfiability queries, which are independent and therefore are subject to parallelization. So, a cluster is used to compute these bounds. We are working on alternative, more efficient, ways of computing tight bounds. In particular, we are currently developing tight bound computation mechanisms that can be run on a single workstation, with an efficiency comparable to the approach in [5], but which may lead to less precise bounds.

We used symbolic execution on the repOK() method, to analyze the effectiveness of using field bounds. This can be used, e.g., to generate all valid structures (within a provided scope), to be employed later on for testing. Moreover, since the repOK() method typically uses all fields of a structure, it does not have any bias towards particular visits of the analyzed structures. A different approach, that we plan to explore, would be to symbolically execute the code under analysis, and then to check which valid structures are required. This would produce structures without necessarily having to instantiate all their parts. The contribution of tight field bounds in such contexts might be different from what we obtained in this work, so we plan to evaluate our approach in such scenarios.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by the Argentinian Ministry of Science and Technology and the South-African Department of Science and Technology, through grant MINCyT-DST SA1108; by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grants PICT PAE 2007 No. 2772 and PICT 2010 No. 1690; and by the MEALS project (EU FP7 programme, grant agreement No. 295261).

References

1. Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, 2008.
2. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.
3. Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society.
4. Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 130–145, Berlin, Heidelberg, 2008. Springer-Verlag.
5. Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 25–36, July 2010.
6. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.
7. Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 14–25, New York, NY, USA, 2000. ACM.
8. Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2619, pages 553–568. Springer, April 2003.
9. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
10. Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.
11. Danhua Shao, Sarfraz Khurshid, and Dewayne E. Perry. Whispec: white-box testing of libraries using declarative specifications. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 11–20, New York, NY, USA, 2007. ACM.
12. Mandana Vaziri and Daniel Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 505–520, Berlin, Heidelberg, 2003. Springer-Verlag.
13. Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 97–107. ACM, 2004.
14. Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for Java containers using state matching. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 37–48. ACM, July 2006.