Efficient Tight Field Bounds Computation Based on Shape Predicates*

Pablo Ponzio[1], Nicolás Rosner[2], Nazareno Aguirre[1,4], and Marcelo Frias[3,4]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Argentina
{pponzio,naguirre}@dc.exa.unrc.edu.ar
[2] Departamento de Computación, FCEFyN, Universidad de Buenos Aires, Buenos Aires, Argentina
nrosner@dc.uba.ar
[3] Departamento de Ingeniería de Software, Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina
mfrias@itba.edu.ar
[4] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract. Tight field bounds contribute to verifying the correctness of object oriented programs in bounded scenarios, by restricting the values that fields can take to feasible cases only, during automated analysis. Tight field bounds are computed from formal class specifications. Their computation is costly, and existing approaches use a cluster of computers to obtain the bounds, from declarative (JML) formal specifications.

In this article we address the question of whether the language in which class specifications are expressed may affect the efficiency with which tight field bounds can be computed. We introduce a novel technique that generates tight field bounds from data structure descriptions provided in terms of shape predicates, expressed using separation logic. Our technique enables us to compute tight field bounds faster on a single workstation, than the alternative approaches which use a cluster, in wall-clock time terms. Although the computed tight bounds differ in the canonical ordering in which data structure nodes are labeled, our computed tight field bounds are also effective. We incorporate the field bounds computed with our technique into a state-of-the-art SAT based analysis tool, and show that, for various case studies, our field bounds allow us to handle scopes in bounded exhaustive analysis comparable to those corresponding to bounds computed with previous techniques.

1 Introduction

Determining to what extent a software artifact is correct is an essential activity in software engineering, and formal methods have contributed with many methodologies and techniques to deal with it. Among these techniques, "push

* This work was partially supported by ANPCyT PICT 2010-1690 and 2012-1298, and by the MEALS project (EU FP7 MEALS - 295261).

© Springer International Publishing Switzerland 2014

button" formal analysis techniques, i.e., those that do not require user intervention, have received special attention. However, automation usually seriously impacts on scalability. In an attempt to deal with scalability issues that typically affect automated analyses, different approaches can be taken in order to simplify or somehow "limit" the software under analysis. Bounded exhaustive verification is one of these approaches, that consists of checking the correctness of a program with respect to its formal specification, but under certain constraints. The approach introduced in [9] is one of the various bounded exhaustive verification settings, in which the number of iterations that loops may perform, as well as the maximum number of objects to be considered for every class involved, are bounded, in order to assess the correctness of an object oriented program with respect to its specification, using SAT solving. This approach has proved to be very useful in finding bugs in object oriented programs [6,13,22,9].

Despite the "limits" imposed on the software under analysis, bounded exhaustive verification still suffers from scalability issues, enabling in many cases to analyze programs only for very small *scopes* (the limit in the number of loop unrolls and maximum number of objects per class). In order to further scale up formal automated analysis, in [9,10] the authors show that by appropriately removing infeasible cases from the values that class fields can take, bounded exhaustive verification can be significantly improved. This mechanism, known as *tight field bound computation*, is used prior to the actual analysis, and has proved to be extremely useful for bounded verification, automated test input generation and for improving symbolic execution [10,1,11].

Tight field bounds depend on a formal specification of the valid inputs of a program under analysis, given in terms of class invariants. Such specification is used to check which field values in the inputs are infeasible (prior to the execution of the program), and therefore can be removed from the representation of the verification problem prior to analysis. To compute field bounds, a large number of feasibility queries have to be performed. The only proposed approach available to effectively compute tight field bounds is introduced in [9]. It is based on declarative formal specifications of class invariants of Java programs given in JML, and requires a cluster for effectively carrying out this task.

In this work, we study whether the efficiency in tight field bound computation depends on the formal language used for expressing class invariants. More precisely, we show that, if the class invariants used for tight field bound computation are expressed using separation logic's inductive *shape predicates* [16], field bounds can be computed efficiently. Although less expressive than JML, shape predicates are expressive enough to describe a broad set of interesting data structures [16], and have been employed as data structure specification language by various tools for software analysis. We introduce a novel approach for tight field bounds computation, which exploits the fact that shape predicates describe linked data structures very precisely, and their inductive definition makes them suitable for tight bounds computation. Furthermore, our field bounds computation approach runs on a single workstation, more efficiently, in wall-clock time terms, than the approach introduced in [9] using a cluster of computers,

```
class AVLTree {                          class Node {
    Node root;                               int data;
}                                            Node left;
                                             Node right;
                                         }
```

Fig. 1. Classes for AVL trees

thus showing that our approach is several orders of magnitude faster. However, the field bounds computed by our approach correspond to a different canonical ordering of structures' nodes, with respect to [9]. Indeed, while [9] canonically orders nodes in a breadth first fashion, which results in smaller scopes for some fields, our approach intrinsically leads to depth first node orderings. To assess the usefulness of the "depth first" field bounds computed by our approach, we incorporate the field bounds into the tool used in [9], and show that they allow us to handle scopes that are comparable to those handled by "breadth first" field bounds, in bounded exhaustive analysis.

2 Bounded Verification and Tight Field Bounds

Tight field bounds help improving various kinds of analysis, one of which is bounded verification via SAT solving, as performed in [9,10]. Therein, a process for verifying whether a given Java program satisfies a JML specification in *bounded* contexts, is presented. This process is based on, given a *scope* (maximum number of loop iterations, and maximum number of instances of the classes involved), encoding the program and its formal specification as a propositional formula, in such a way that the resulting formula is satisfiable if and only if there exists an execution of the program within the provided scope that violates the specification. If the resulting formula is unsatisfiable, then the program is correct with respect to its specification, within the provided scope.

Various intermediate languages are employed in [9,10] during the translation from Java code and JML contracts to a propositional formula. In particular, the relational formal languages DynAlloy [8], Alloy [14] and KodKod [20] are involved in the process. KodKod [20] is able to profit from *upper bounds* for relations. These bounds capture information about which tuples in the relations involved in a relational constraint problem (in our case, resulting from the translation of an annotated program) are *infeasible* due to the constraints. Since tuples in the domains of relations are represented as propositional variables in the formula resulting from the translation, infeasible cases lead to removing the corresponding variables (or, more precisely, replacing the corresponding variables by *false*). This is highly relevant, since variable removal contributes to scaling up SAT based analysis.

In order to introduce the concept of tight field bound, let us first describe briefly, by means of an example, the intermediate representation of the Java heap in Alloy and KodKod. For further details, we refer the reader to [9]. Consider the classes in Fig. 1, which may be part of a definition of AVL trees. In (Dyn)Alloy, program states are captured by sets of object identifiers to represent class extensions, and binary relations from the class extension to the corresponding datatype, to represent fields. For instance, for the classes in Fig. 1, a program state would comprise sets $AVLTree$ and $Node$, and relations $root \subseteq AVLTree \times (Node \cup \{null\})$ and $left$, $right \subseteq Node \times (Node \cup \{null\})$ (we disregard integer fields for presentation purposes). Assuming scope 3 for `Node` in the analysis, class `Node` is represented in KodKod as the set $Node = \{N_0, N_1, N_2\}$, while field `left` is represented by a relation $left \subseteq \{N_0, N_1, N_2\} \times \{N_0, N_1, N_2, null\}$.

In the translation from a relational model to a propositional formula, relations are represented via sets of propositional variables. For instance, relation *left* above is represented by propositional variables:

$$\{l_{x,y} \mid x \in \{N_0, N_1, N_2\} \wedge y \in \{N_0, N_1, N_2, N_3, null\}\}.$$

The variables in this set capture the possible values for field `left`, in the corresponding program state. More precisely, a variable l_{N_i,N_j} being true in a given instance of a constraint solving problem indicates that nodes N_i and N_j are related via field `left` in the corresponding program state.

For example, assuming similar representations for fields `root` and `right`, when variables ro_{T_0,N_0} (for root), l_{N_0,N_1}, $l_{N_1,null}$, $l_{N_2,null}$ (for left), r_{N_0,N_2}, $r_{N_1,null}$ and $r_{N_2,null}$ (for right) are true, and all the remaining variables are false, we obtain the structure of Fig. 2(a).

Notice that constraints that are part of the specification force some variables in the resulting relational constraint problem to be false. For instance, if the linked structure is acyclic in a given state (e.g., in the state prior to the execution of the program under analysis), variables l_{N_i,N_i} are all necessarily false. Thus, these variables can be replaced by false, reducing the number of variables required to encode bounded program correctness for SAT solving. More precisely, if we know beforehand that certain relationships between heap objects are forbidden, we can remove the infeasible variables that represent them. KodKod allows one to do so, by providing an *upper bound*. Formally, an upper bound for a field $f : A \to B$ is a relation $U_f \subseteq A \times B$. Given an upper bound U_f, KodKod will get rid of all the variables $p_{a,b} \in M_f$ such that $(a, b) \notin U_f$, replacing them by false.

Of course, the "tighter" the upper bound, the better, since tighter bounds allow one to remove more variables (recall that SAT solving algorithms have an exponential worst case time complexity, that depends on the number of propositional variables). However, we are interested in considering only *sound* upper bounds, i.e., those composed solely by infeasible variables, otherwise we would compromise the whole SAT-based analysis. For instance, an upper bound

$$U_{left\_wrong} = Node \times (Node \cup null) - \{(N_0, null), (N_1, null), (N_2, null)\}$$

forbids the `left` field of any node to be null, causing the analysis to omit all the valid non-empty AVL tree instances. Thus, we want to compute tight bounds that only get rid of infeasible variables in the propositional formula encoding a program state.

Tight field bounds are useful for analysis. However, determining these bounds is not easy, and they have to be computed from specifications, prior to analysis. In particular, in [9,10] tight field bounds are computed from declarative JML specifications, which are translated into Alloy's relational logic. As an example, consider the following fragment of a relational logic specification of AVL trees:

```
AVL_Invariant:
(all n: Node | n in this.root.*(left + right) - null =>
               n !in n.^(left + right)) and
  ...
```

This fragment specifies acyclicity, using closure operators (`*` is reflexive-transitive closure, while `^` is transitive closure). These specifications are complemented with symmetry-breaking predicates, which are automatically produced from class specifications [9]. Such symmetry breaking predicates force a canonical, breadth-first ordering, in the labeling of structures' nodes. This helps removing redundant structures (similar to partial-order reduction in the context of model checking). For AVL tree specifications, for instance, the corresponding symmetry breaking predicate would forbid producing the structure in Figure 2(b), while accepting structure in Figure 2(a). Notice that these two structures are isomorphic, and thus is sufficient to consider only one of them (especially in languages like Java, with no pointer arithmetic, where the specific memory addresses where nodes are allocated, abstracted as node labels in this formal representation, is irrelevant). Using class specifications and symmetry breaking, in [9] a tight bound for a field f is computed by querying the SAT solver about the feasibility of each variable in the representation of f. So, for instance, for every pair of nodes N_i and N_j within a given scope, one would have to check:

$$SAT(\texttt{AVL\_Invariant} \wedge \texttt{AVL\_Symmetry\_Breaking} \wedge N_i.\texttt{left}.N_j),$$

that is, is there a (valid) AVL tree within the given scope in which N_j is the left node of N_i? If this is not the case, then the propositional variable representing $N_i.\texttt{left}.N_j$ can be removed. All these queries are independent, and therefore can be performed in parallel. The actual process for computing tight bounds in [9] uses an *iterative* approach, that first removes variables whose infeasibility can be quickly determined (and maintaining those whose feasibility is determined). Those that reach a timeout are processed again, after simplifying the satisfiability problem thanks to the variables already determined infeasible. In order to carry out this process effectively, a cluster of computers is employed [9,10].

2.1 Tight Bounds and Separation Logic Invariants

Separation logic [19] is an extension of first order logic that enables one to reason about programs dealing with heap allocated data structures in a concise manner. It provides two novel operators to describe heap properties: separating conjunction ($*$), and separating implication ($-\!\!*$). Intuitively, $h_1 * h_2$ describes a heap that comprises two disjoint parts satisfying formulas h_1 and h_2, respectively. We

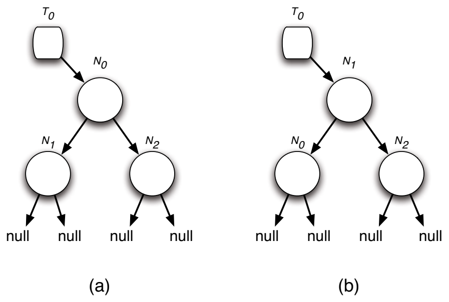


Fig. 2. Two isomorphic AVL tree instances

$$avl(t_0, h_0) \doteq (t_0 = null \wedge h_0 = 0) \vee (t_0 \mapsto t_1, t_2 * avl(t_1, h_1) * avl(t_2, h_2) \wr \\ h_0 = 1 + max(h_1, h_2) \wedge |h_1 - h_2| \leq 1)$$

Fig. 3. AVL tree specification given as a shape predicate

do not consider $-\!\!*$ in this work; we refer the reader to [19] for details. In separation logic, inductive *shape predicates* are used to describe heap allocated data structures, as well and their state evolution as a program is executed. Figure 3 shows a sample shape predicate characterizing AVLs. Symbol $\wr$ separates the spacial part from the pure part of a shape predicate. It represents a conjunction, since a predicate is satisfied if both the spatial and pure parts are satisfied.

In this paper we study if tight field bounds can be computed more efficiently, if class invariants are expressed in a different formal language. We propose expressing such predicates in separation logic. Separation logic inductive shape predicates provide useful information, that can be exploited to efficiently compute field bounds. Consider for instance the shape predicate in Fig. 3, characterizing AVLs. Notice that whenever $avl(n, h)$ and $n \neq null$, we know that $n.left \neq n$, since $*$ forces the "left subtree" of n to be in a disjoint part of the heap with respect to n. Furthermore, a particular unfolding of shape predicate p univocally denotes a shape of the data structure defined by p (if the accumulated pure part is satisfiable). For example, unfolding the *avl* predicate (Fig. 3) as follows (we disregard the height in this unfolding, for the sake of simplicity):

$$t_0 \mapsto t_1, t_2 * t_1 \mapsto t_3, t_4 * t_2 \mapsto t_5, t_6 \wr \\ t_3 = null \wedge t_3 = null \wedge t_4 = null \wedge t_5 = null \wedge t_6 = null$$

Due to the semantics of $*$, variables t_0, t_1 and t_2 must be replaced by different node identifiers N_0, N_1 and N_2, respectively. We thus obtain the shape of the AVL in Fig 2(a). Assuming that *avl* is our class invariant, a tight bound for field `left` is forced to contain pairs (N_0, N_1), $(N_1, null)$, $(N_2, null)$ (similarly for `right`), since otherwise the valid shape of Fig. 2(a) would be disallowed, and the analysis would not be sound.

$$\begin{array}{lll}
p(v^*) := & \bigvee(\exists v'^* : \Gamma \wr \Sigma) \\
\Gamma & := & emp \mid v_k \mapsto v_{k_1}, .., v_{k_n} | p(v^*) | \Gamma_1 * \Gamma_2 \\
\Sigma & := & \gamma \wedge \phi \\
\gamma & := & v_1 = v_2 \mid v = null \mid v_1 \neq v_2 \mid v \neq null \mid \gamma_1 \wedge \gamma_2 \\
\phi & := & b \mid a \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \exists v : \phi \mid \forall v : \phi \\
b & := & true \mid false \mid v \mid b_1 = b_2 \\
a & := & s_1 = s_2 \mid s_1 \leq s_2 \\
s & := & k^{int} \mid v \mid k^{int} \times s \mid s_1 + s_2 \mid -s \mid max(s_1, s_2) \mid min(s_1, s_2)
\end{array}$$

Fig. 4. Shape predicate specification framework ([16])

Our approach is based on the above described observations. Intuitively, given a shape predicate p and a finite sequence of node identifiers as input, our approach works by recursively unfolding p, canonically labeling nodes, and adding the corresponding pairs to the resulting tight bound. When all the structures comprising the input node sequence have been "visited", the approach finishes returning a tight bound.

We will consider shape predicates defined using the specification framework of [16], shown in Figure 4 (with slightly modified syntax). The framework supports shape predicates encompassing a spatial and a pure part (Γ and Σ, respectively). The spatial part is a sequence of $*$ separated formulas describing how a data structure is organized in the heap (γ). The pure part is heap independent, and, in [16], is restricted to pointer equality/inequality (γ) and Presburger arithmetic (ϕ). As it will be discussed later on, our analysis supports more expressive shape predicates, e.g., allowing Σ to be an arbitrary arithmetic formula. However, for illustration purposes, we restrict ourselves to the framework above, and regard richer extensions as future work. For the sake of clarity, we assume all free variables of shape predicates to be existentially quantified (we therefore omit existential quantifiers).

It is worth mentioning that shape predicates are less expressive than JML. JML allows one to describe structures that "share" substructures of the heap, some of which cannot be captured by our employed shape predicate specification framework. However, as we mentioned, shape predicates are very expressive, being able to capture many heap-allocated datatypes, and as we show in this paper, enabling us to compute tight bounds very efficiently, contributing to the SAT based analysis of these structures.

3 Tight Bounds Calculation from Shape Predicates

For the sake of clarity, we will start by describing how tight field bounds can be computed from shape predicates using a brute force technique. We will then explain how this starting technique is improved, both in terms of memory consumption, and computation time, in particular by normalizing the inputs of the

Algorithm 1. UNFOLD algorithm

```
1: function UNFOLD($p(r, v_1, \ldots), f, l$)
2:     if $l - f = 0$ then                                        ▷ No addresses
3:         let $base(p) = emp \wr bt_1 \wedge \ldots \wedge bt_j$ return $\{base(p)\}$
4:     $result = \emptyset$                    ▷ There are addresses available ($l - f > 0$)
5:     let $ind(p(r, \ldots)) = r \mapsto r_1, \ldots * ip_1(y_1, \ldots) * \cdots * ip_i(y_i, \ldots) \wr it_1 \wedge \ldots \wedge it_l$
6:         ▷ Share $l - (f + 1)$ addresses between recursive calls
7:         for $(f_1, l_1), .., (f_i, l_i) \in partition(f + 1, l, i)$ do
8:             $set_1 =$UNFOLD$(ip_1(y_1, \ldots), f_1, l_1)$
9:             ...
10:            $set_i =$UNFOLD$(ip_i(y_i, \ldots), f_i, l_i)$
11:            for $(s_1, \ldots, s_i) \in set_1 \times \ldots \times set_i$ do
12:                $result = result \cup \{(r \mapsto r_1, \ldots \wr it_1 \wedge \ldots \wedge it_l) \uplus s_1 \uplus \ldots \uplus s_i\}$
13:        return $result$
```

brute force algorithm, and applying memoization. For presentation reasons, we describe our technique on a subset of the shape predicates definable in the framework of Section 2.1, namely predicates with one base and one inductive case. Extending the technique to support more general shape predicates is straightforward (although we do not deal with this extension, due to space reasons). Without loss of generality, we assume that the only variable allowed to be bound to a heap node in the right-hand side of a shape predicate is r (short for root). In summary, throughout this section we consider shape predicates to have the form:

$$\begin{aligned} p(r, v_1, \ldots) =& (emp \wr bt_1 \wedge \ldots \wedge bt_j) \vee \\ & (r \mapsto r_1, \ldots * ip_1(y_1, \ldots) * \cdots * ip_i(y_i, \ldots) \wr it_1 \wedge \ldots \wedge it_l) \end{aligned}$$

where $bt_z, 1 \leq z \leq j$, $it_z, 1 \leq z \leq l$ are pure terms (cf. Section 2.1), and $ip_z, 1 \leq z \leq i$ are shape predicate's names (possibly distinct from p). Finally, we assume a fixed set of fields $f_1, .., f_n$ for the heap, and an ordered set of addresses $A = [N_0, N_1, ...]$, which corresponds to the allowed labels for reference fields.

Let us start describing the brute force approach. The inputs to this algorithm are a shape predicate p describing the valid instances of the heap, and the indexes f and l of an ordered subset $[N_f, ..., N_l]$ of A. Its outputs are tight field bounds for fields $f_1, \ldots, f_n$, for heaps with exactly $l - f$ input nodes (labeled $N_f, \ldots, N_{l-1}$). The brute force approach is composed of various stages, namely, predicate unfolding, concrete instance generation, and tight bound construction. We now describe these stages in detail.

Unfolding shape predicates. The brute force approach starts by unfolding a predicate, as indicated by Function UNFOLD shown as Algorithm 1. UNFOLD(p, f, l) yields the set of separation logic formulas representing instances of p with exactly $l - f$ nodes.

As an example, consider the shape predicate for AVLs, shown in Fig. 3. UNFOLD$(avl(r_0, h_0), 0, 2)$ should return two separation logic formulas, representing the two feasible AVL's with two nodes. When executing UNFOLD($avl(r_0, h_0), 0, 2$), in line 7, $partition(0+1, 2, 2)$ tries all the feasible partitions of the sequence $[N_1]$ of addresses (corresponding to the interval $f = 1, l = 2$) between the two recursive calls. It returns two possibilities: assigning node N_1 to the first recursive call (indexes $(1, 2), (2, 2)$ in the main loop) and none to the second, or passing no nodes to the first recursive call and assigning N_1 to the second (indexes $(1, 1), (1, 2)$). Let us consider the first case in more detail. The first recursive call, corresponding to UNFOLD($avl(r_1, h_1), 1, 2)$), yields $\{t_1 \mapsto h_1, t_2, t_3 * emp * emp \wr h_1 = 1 + m(h_2, h_3) \wedge a(h_2 - h_3) \leq 1 \wedge r_2 = r_3 = null \wedge h_2 = h_3 = 0\}$ (one formula standing for a tree with exactly one node), whereas the second, UNFOLD($avl(r_2, h_2), 2, 2$) produces $\{emp \wr r_2 = null \wedge h_2 = 0\}$ (one formula representing the empty tree). The inner loop, line 11, iterates over all the feasible combinations of formulas for the left and right trees, i.e., formulas in the cartesian product of the sets resulting from the i recursive calls. Our running example has only one possible combination, as the results of the recursive calls were singletons. Then, line 12 combines the formula standing for the root of the structure with each of the feasible pairs of left and right subtrees. In our example, this is $r_0 \mapsto h_0, r_1, r_2$, $t_1 \mapsto h_1, t_2, t_3 * emp * emp \wr h_1 = 1 + m(h_2, h_3) \wedge a(h_2 - h_3) \leq 1 \wedge r_2 = r_3 = null \wedge h_2 = h_3 = 0$ and $emp \wr r_2 = null \wedge h_2 = 0$, respectively. In this step, the algorithm uses operator $\uplus$, which merges the spatial and pure parts of its input formulas using $*$ and $\wedge$, respectively.

In this example, UNFOLD($avl(r_0, h_0), 0, 2$) leads to the following pair of formulas:

$$\begin{array}{l|l}
t_0 \mapsto h_0, t_1, t_4 * t_1 \mapsto h_1, t_2, t_3 * emp \wr & t_0 \mapsto h_0, t_1, t_2 * emp * t_2 \mapsto h_2, t_3, t_4 \wr \\
h_0 = 1 + m(h_1, h_4) \wedge a(h_1 - h_4) \leq 1 \wedge & h_0 = 1 + m(h_1, h_2) \wedge a(h_1 - h_2) \leq 1 \wedge \\
\quad h_1 = 1 + m(h_2, h_3) \wedge a(h_2 - h_3) \leq 1 \wedge & \quad h_2 = 1 + m(h_3, h_4) \wedge a(h_3 - h_4) \leq 1 \wedge \\
\quad t_2 = t_3 = t_4 = null \wedge & \quad t_1 = t_3 = t_4 = null \wedge \\
\quad h_2 = h_3 = h_4 = 0 & \quad h_1 = h_3 = h_4 = 0
\end{array}$$

These formulas stand for all the feasible AVL instances with exactly two nodes. It is worth noticing again that, due to the semantics of operator $*$, in line 7 the algorithm can feed the root node and its recursive calls with disjoint partitions of the input address set (the domains of the subheaps $r_0 \mapsto h_0, r_1, r_2$, $avl(r_1, h_1)$, and $avl(r_2, h_2)$ must be all disjoint in $r_0 \mapsto h_0, r_1, r_2 * avl(r_1, h_1) * avl(r_2, h_2)$). Thus, this allows us to ignore many distributions of addresses to subheaps that involve aliasing.

Generating concrete instances from separation logic formulas. The second step produces all the concrete heap instances represented by the formulas returned by UNFOLD. Notice that (as seen in the example of the previous section) each formula f returned by UNFOLD comprises a pure part: $pr_f = t_1 \wedge \ldots \wedge t_l$, and a spatial part: $sp_f = x_1 \mapsto x_{1,1}, .. * \ldots * x_m \mapsto x_{m,1}, \ldots$. First, we perform a symmetry breaking procedure in order to reduce the number of feasible instances of a formula f yielded by UNFOLD. This procedure involves traversing sp_f from left to right, assigning address N_i to each variable x_i such that $x_i \mapsto x_{i,1}, .. \in sp_f$ during the traversal. Applying this procedure to the instances in the example above yields (formulas in boldface were added in this step):

$$\begin{array}{l|l}
t_0 \mapsto h_0, t_1, t_4 * t_1 \mapsto h_1, t_2, t_3 * emp \wr & t_0 \mapsto h_0, t_1, t_2 * emp * t_2 \mapsto h_2, t_3, t_4 \wr \\
h_0 = 1 + m(h_1, h_4) \wedge a(h_1 - h_4) \leq 1 \wedge & h_0 = 1 + m(h_1, h_2) \wedge a(h_1 - h_2) \leq 1 \wedge \\
\quad h_1 = 1 + m(h_2, h_3) \wedge a(h_2 - h_3) \leq 1 \wedge & \quad h_2 = 1 + m(h_3, h_4) \wedge a(h_3 - h_4) \leq 1 \wedge \\
\quad t_2 = t_3 = t_4 = null \wedge & \quad t_1 = t_3 = t_4 = null \wedge \\
\quad h_2 = h_3 = h_4 = 0 \wedge & \quad h_1 = h_3 = h_4 = 0 \wedge \\
\quad \mathbf{t_0 = N_0 \wedge t_1 = N_1} & \quad \mathbf{t_0 = N_0 \wedge t_2 = N_1}
\end{array}$$

The soundness of this procedure is guaranteed by the semantics of the $*$ operator. Notice that this fixes the ordering of addresses in valid heap instances, and therefore induces a heap canonicalization. Thus, it can be thought of as the equivalent of using symmetry breaking predicates in TACO's tight bound computation procedure.

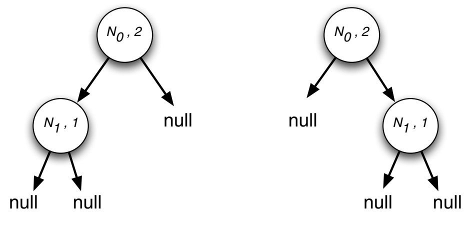
Next, we invoke a decision procedure in order to obtain the models of the formulas generated in the previous step, i.e, to yield concrete structures. Observe that all the variables in the formulas are existentially quantified, and their pure part comprises only conjunctions of formulas in the language of Section 2.1. Therefore, we can encode the pure part of each formula produced in the previous step in the input language of any modern SMT solver, to obtain concrete instances from it. Continuing with our example, after calling a decision procedure with the formulas we get:

$$\begin{array}{l|l}
t_0 \mapsto h_0, t_1, t_4 * t_1 \mapsto h_1, t_2, t_3 * emp \wr & t_0 \mapsto h_0, t_1, t_2 * emp * t_2 \mapsto h_2, t_3, t_4 \wr \\
\mathbf{h_0 = 2 \wedge h_1 = 1} \wedge & \mathbf{h_0 = 2 \wedge h_2 = 1} \wedge \\
\quad t_2 = t_3 = t_4 = null \wedge & \quad t_1 = t_3 = t_4 = null \wedge \\
\quad h_2 = h_3 = h_4 = 0 \wedge & \quad h_1 = h_3 = h_4 = 0 \wedge \\
\quad t_0 = N_0 \wedge t_1 = N_1 & \quad t_0 = N_0 \wedge t_2 = N_1
\end{array}$$

In this case, both formulas are satisfiable and have one model. Therefore, each of these formulas represents a valid heap instance, which can be obtained by replacing variables by values in the formula's spatial part:

$$N_0 \mapsto 2, N_1, null * (N_1 \mapsto 1, null, null) * (emp)$$
$$N_0 \mapsto 2, null, N_1 * (emp) * (N_1 \mapsto 1, null, null)$$

Graphically, these formulas correspond to the following tree structures:



Tight field bounds from concrete heap instances. The last step uses the heap instances produced by the previous step to build tight field bounds. The resulting tight field bound is composed by the union of all the field values occuring in any of the aforementioned heap instances. Returning to our example, the field values added by each of the instances above are:

$$\begin{array}{r|l}
left = N_0 \mapsto N_1 + N_1 \mapsto null & left = N_0 \mapsto null + N_1 \mapsto null \\
right = N_0 \mapsto null + N_1 \mapsto null & right = N_0 \mapsto N_1 + N_1 \mapsto null \\
height = N_0 \mapsto 2 + N_1 \mapsto 1 & \quad height = N_0 \mapsto 2 + N_1 \mapsto 1
\end{array}$$

The union of the values above results in the AVL bounds for scope exactly 2:

$left = N_0 \mapsto N_1 + N_0 \mapsto null + N_1 \mapsto null$
$right = N_0 \mapsto N_1 + N_0 \mapsto null + N_1 \mapsto null$
$height = N_0 \mapsto 2 + N_1 \mapsto 1$

It is important to remark that SAT-based analyses typically use non-strict scopes. That is, if scope k is given, the analysis explores all the feasible heap instances with up to k nodes. To compute tight field bounds in such a case, we sequentially run the brute force approach for up to k nodes, and return the union of the resulting bounds.

3.1 Improvements to the Brute Force Algorithm

The approach just introduced must generate a potentially exponential number of structures before computing tight field bounds. To avoid this problem, we can compute bounds on the fly, during the traversal of the input shape predicate, without generating instances. This leads to an alternative algorithm SLFieldBounds$(p(r, v_1, ..., v_n), f, l)$, which produces as output a pair containing: *(i)* a tight field bound for scope exactly $l - f$, for the structure defined by p, and *(ii)* the set of feasible assignments of values to the variables $r, v_1, ..., v_n$.

Let us illustrate this alternative with an example. Suppose we execute SLFieldBounds$(avl(r_0, h_0), 0, 5)$ to obtain field bounds for scope exactly 5, for AVLs. SLFieldBounds traverses the input shape predicate in the same way as Unfold. The computation of field bounds for AVLs with exactly 5 nodes involves at some point recursive calls SLFieldBounds$(avl(r_1, h_1), 1, 3)$ and SLFieldBounds$(avl(r_2, h_2), 3, 5)$. The following table shows the results of these calls:

parameters	bound	assignment
$avl(r_1, h_1)$ $[N_1, N_2]$	$l : N_1 \mapsto N_2 + N_1 \mapsto null + N_2 \mapsto null$ $r : N_1 \mapsto N_2 + N_1 \mapsto null + N_2 \mapsto null$ $h : N_1 \mapsto 2 + N_2 \mapsto 1$	$\{(r_1 = N_1, h_1 = 2)\}$
$avl(r_2, h_2)$ $[N_3, N_4]$	$l : N_3 \mapsto N_4 + N_3 \mapsto null + N_4 \mapsto null$ $r : N_3 \mapsto N_4 + N_3 \mapsto null + N_4 \mapsto null$ $h : N_3 \mapsto 2 + N_4 \mapsto 1$	$\{(r_2 = N_3, h_2 = 2)\}$

At this point, the algorithm assigns N_0 to the root node, pointed to by variable r_0, and builds a bound for the combination of the root node $(r_0 \mapsto h_0, r_1, r_2)$ with the results of the recursive calls shown above. However, the algorithm only adds field values to the resulting bound when the pure part of the predicate is satisfiable, otherwise field values of invalid structures would be included in the resulting bound. That is, we incorporate the constraint solving that in the brute force takes place when building concrete instances, during the process of unfolding and traversing the formula. For the example, the formula to solve is: $h_0 = 1 + m(h_1, h_2) \wedge a(h_1, h_2) \leq 1 \wedge r_1 = N_1 \wedge h_1 = 2 \wedge r_2 = N_3 \wedge h_2 = 2 \wedge r_0 = N_0$ which has only one model: $r_0 = N_0 \wedge r_1 = N_1 \wedge r_2 = N_3 \wedge h_0 = 3 \wedge \ldots$.

Thus, SLFieldBounds performs a search for all the models of the pure part of its input predicate, using all the feasible combinations of assignments in recursive calls together with the assignment for the root node.

Table 1. Times required for computing tight field bounds from JML specifications and from shape predicates, for various case studies (in MM:SS.sss)

		S5	S7	S10	S12	S15	S17
Linked List	TACO(w)	00:11	00:11	00:15	00:24	00:47	01:04
	TACO(seq)	02:56	02:56	04:00	06:24	12:32	17:04
	SL	00:00.065	00:00.091	00:00.094	00:00.110	00:00.129	00:00.141
BSTree	TACO(w)	00:11	00:11	00:16	00:38	01:56	04:05
	TACO(seq)	02:56	02:56	04:16	10:08	30:56	65:20
	SL	00:00.268	00:00.209	00:00.508	00:00.633	00:01.032	00:01.389
TreeSet	TACO(w)	00:16	00:30	01:44	02:51	05:19	16:42
	TACO(seq)	04:16	08:00	27:44	45:36	85:04	267:12
	SL	00:00.667	00:01.159	00:02.693	00:04.352	00:08.266	00:11.483
AVL	TACO(w)	00:17	00:32	01:55	03:46	10:36	47:25
	TACO(seq)	04:32	08:32	30:40	60:16	169:36	2845:00
	SL	00:00.121	00:00.195	00:00.403	00:00.561	00:00.944	00:01.387

Memoization. When fed with shape predicates with more than one recursive call, as in the case of AVLs, cases can be repeated. As an example, we showed above calls SLFieldBounds$(avl(r_1, h_1), 1, 3)$ and SLFieldBounds$(avl(r_2, h_2), 3, 5)$. Notice that the resulting bounds are tight field bounds for AVLs with exactly two consecutive addresses. They are equivalent, up to renaming. SLFieldBounds solves these equivalent problems independently, and thus repeats computations. To avoid this problem, we "memoize" SLFieldBounds, i.e., we use a matrix M to store the results of calls to SLFieldBounds, "normalized" to have 0 as a starting address. Whenever a call to SLFieldBounds is made with a given scope (the pair f and l), we check first whether the corresponding "normalized" bound and assignment have been computed before, to avoid recomputing it. Of course, to produce the actual bound, a shift must be applied to the normalized stored bounds and assignments, before returning it.

4 Experimental Results

To assess our approach, we perform two experiments. The first is a comparison of our algorithm for tight bounds computation from shape predicates, with `TACO`, that computes tight bounds from JML specifications. The second is an assessment of the profit provided by the tight bounds computed by our algorithm, that as was mentioned before, differ in some cases from those computed by `TACO`, since the canonical ordering of nodes is different (depth first in our case, vs. breadth first in the case of `TACO`). For both assessments, we consider the following data structures: an implementation of sequences based on singly linked lists (Linked List); `TreeSet` from package `java.util`, based on red-black trees (TreeSet); AVL trees from [2] (AVL); and binomial heaps used in [21] (Binomial Heap). These classes have JML specifications, used in [9] for analysis.

Table 2. Comparison of the impact of "depth first" vs. "breadth first" tight field bounds, in SAT-based bounded verification (in H:MM:SS)

			S5	S7	S10	S12	S15	S17
Linked List	Contains	`TACO`	0:00:01	0:00:02	0:00:03	0:00:04	0:00:05	0:00:07
		`SL`	0:00:01	0:00:02	0:00:03	0:00:04	0:00:05	0:00:07
	Insert	`TACO`	0:00:02	0:00:02	0:00:03	0:00:04	0:00:05	0:00:09
		`SL`	0:00:02	0:00:02	0:00:03	0:00:04	0:00:05	0:00:09
	Remove	`TACO`	0:00:02	0:00:02	0:00:04	0:00:06	0:00:07	0:00:13
		`SL`	0:00:02	0:00:02	0:00:04	0:00:06	0:00:07	0:00:13
BSTree	Find	`TACO`	0:00:02	0:00:12	0:26:15	TO	TO	TO
		`SL`	0:00:02	0:00:16	0:24:20	2:48:21	TO	TO
	Remove	`TACO`	0:00:02	0:00:09	0:08:35	2:06:14	TO	TO
		`SL`	0:00:02	0:00:08	0:05:43	1:39:50	TO	TO
	Insert	`TACO`	0:02:06	0:29:22	TO	TO	TO	TO
		`SL`	0:02:15	0:33:15	TO	TO	TO	TO
AVL	Find	`TACO`	0:00:03	0:00:05	0:00:16	0:00:36	0:02:44	0:12:36
		`SL`	0:00:03	0:00:05	0:00:31	0:01:09	0:15:43	0:35:27
	FindMax	`TACO`	0:00:01	0:00:01	0:00:02	0:00:03	0:00:05	0:00:08
		`SL`	0:00:01	0:00:01	0:00:02	0:00:04	0:00:09	0:00:31
	Insert	`TACO`	0:00:56	0:01:03	0:03:42	0:11:48	1:15:15	TO
		`SL`	0:00:57	0:01:05	0:04:25	0:14:12	1:12:32	TO
TreeSet	Find	`TACO`	0:00:03	0:00:05	0:00:47	0:02:19	0:16:45	1:15:05
		`SL`	0:00:03	0:00:07	0:00:48	0:03:47	0:34:04	1:31:50
	Insert	`TACO`	0:00:30	0:02:54	TO	TO	TO	TO
		`SL`	0:00:29	0:02:34	TO	TO	TO	TO

We took shape predicates characterizing these structures from the literature, and performed slight modifications to make them equivalent to the corresponding JML descriptions. We ran our algorithm on a 2.0Ghz, 2MB cache computer, with a dual core processor. `TACO` was run on a cluster of 16 computers, each with 2 dual core processors (2.67GHz, 2MB cache per core) each (as reported in [9]). Running times are reported using a format given in the tables' caption, with TO and OoM indicating that the analysis exhausted the time (3 hours) and memory (2Gb), respectively. The experiments can be reproduced by downloading `http://dc.exa.unrc.edu.ar/staff/pponzio/sltb/FM14exp.tgz`, and following the instructions therein.

The results of the comparison of tight bound computations are summarized in Table 1. Our algorithm always terminated in at most a few seconds (time is reported in these cases including milliseconds). The time employed by `TACO` in computing the bounds is shown as wall clock time (TACO(w)), and is also "sequentialized" (TACO(seq)), i.e., an estimation of the sequential time is given, by multiplying the wall clock time by 16, the number of computers. Notice that this estimation is very conservative, since the parallelisation takes advantage of cores (the number of cores is 64), and we are multiplying the parallel time by the number of computers (16). As these experiments show, computing bounds from separation logic shape predicates, our approach, is various orders of magnitude more efficient than doing so from JML specifications.

Our second set of experiments compares the impact of "depth first" tight field bounds, computed from shape predicates, with "breadth first" tight field bounds, computed from JML specifications, in SAT-based bounded verification. We performed bounded verification of several methods of the studied classes, comparing TACO tight bounds with tight bounds computed with our approach. All the analyzed methods are correct with respect to their specifications. We chose correct implementations because they represent the worst case for SAT-based analyses, as they require the whole state space to be explored. Loops were unrolled enough to cover the corresponding scopes, and method calls were inlined. The results are summarized in Table 2. Notice that for lists, both approaches led to the same times. This is so because, in this data structure, breadth first and depth first node labelings coincide, and therefore the field bounds computed with both approaches are the same. For tree-like structures, our "depth first" field bounds yield running times that are similar to those obtained by using the "breadth first" field bounds. In general, the differences are not significant, with depth first bounds being slightly less efficient, with a few exceptions in which our bounds lead to better running times.

5 Related Work

This is the first attempt to use shape predicates for tight bounds calculation. A closely related approach to the work on this paper is that of [9,10], which introduces an algorithm for tight bounds calculation that works by making a big number of parallel queries to SAT solving. The drawback of this approach is that it requires a lot of computational resources. Our experiments show that this algorithm can run for more than an hour on a cluster of sixteen machines (64 cores). In contrast, the approach presented here is significantly more efficient for bounds computation. Our case studies point out that our approach can perform bounds calculation in at most a few seconds, running on a single computer. However, TACO allows for JML specifications of data structures, which can be argued to be easier to write for an average programmer than the shape predicates required by our approach (as well as more expressive). In both works, the calculated bounds are shown useful to achieve better runtime efficiency, in the context of SAT-based bounded exhaustive analysis of Java container classes with rich structural invariants. JForge [6] also performs SAT based analysis of Java programs with JML specifications. We do not provide an experimental comparison with JForge, since the tool has been shown to perform poorly compared to tight bounds based approaches in [9], and has not been improved lately. Further examples of SAT based bounded analyses are presented in [13,22]. They are tailored for C programs, and do not make use of tight bounds. Besides TACO, there are other approaches that benefit from the use of tight bounds. [11] introduces an adaptation of the Lazy Initialization mechanism of Symbolic Java PathFinder, that makes use of tight bounds. [18] introduces a dataflow analysis that allows propagating tight bounds to all the states of a program, starting with bounds for the initial state (as those calculated by TACO and our approach). FAJITA [1] is a version of TACO especially tailored for automated test generation. As is the case with TACO, FAJITA's efficiency heavily relies on tight bounds. All these approaches make use of bounds, but none proposes alternative ways of computing bounds.

We borrow the shape specification mechanism of [16], which we use to capture class invariants, employed as inputs by our algorithms for tight bounds calculation. The traditional use of shape predicates is in the verification of shape and size properties of programs manipulating linked data structures [16]. Other works extend [16] (e.g., [17,7]) by improving the verification process with different mechanisms. In these cases, the focus is on using shape predicates and a corresponding calculus to prove properties of programs via some form of symbolic execution using shape predicates. Other examples of successful separation logic based verification approaches are presented in [3,5,15]; they are concerned with proving memory safety properties of programs. Our approach is different: we use shape predicates to compute bounds, which can then be used for a number of different bounded SAT based analyses, such as bounded verification [9] and test generation [11,1].

6 Conclusion

The use of tight bounds is crucial for improving the efficiency and increasing the scalability of SAT-based bounded verification [9], as well as other related analysis techniques, such as test generation [1] and symbolic execution [11]. In this article we introduced an algorithm for tight bounds calculation based on shape predicates. This algorithm exploits the precision of shape predicates in the description of linked structures, to efficiently compute tight bounds, significantly outperforming TACO, the existing approach to bounds calculation, by several orders of magnitude. Our approach computes field bounds that differ from those computed by TACO, since the canonical ordering considered for the nodes of the structure under analysis is depth first, as opposed to TACO's breadth first labeling. Although this has an impact in the size of bounds for some fields, we showed in our experiments that our bounds are also effective.

Tight bounds have the potential of improving analysis times in other contexts. In this respect, we are working on adapting Korat [4] to use tight bounds for faster test generation. Also, in the context of SAT based white box test generation, we plan to extend path conditions with shape information, using it to remove irrelevant variables from the encoding of traces.

References

1. Abad, P., Aguirre, N., Bengolea, V., Ciolek, D., Frias, M., Galeotti, J., Maibaum, T., Moscato, M., Rosner, N., Vissani, I.: Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving. In: ICST 2013 (2013)

2. Belt, J., Robby, Deng, X.: Sireum/Topi LDP: A Lightweight Semi-Decision Procedure for Optimizing Symbolic Execution-based Analyses. In: FSE 2009 (2009)
3. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011)
4. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: ISSTA 2002, pp. 123–133 (2002)
5. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL 2009 (2009)
6. Dennis, G., Chang, F., Jackson, D.: Verification of Code with SAT. In: ISSTA 2006 (2006)
7. Chin, W.-N., Gherghina, C., Voicu, R., Le, Q.L., Craciun, F., Qin, S.: A specialization calculus for pruning disjunctive predicates to support verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 293–309. Springer, Heidelberg (2011)
8. Frias, M., Galeotti, J., López Pombo, C., Aguirre, N.: DynAlloy: Upgrading Alloy with Actions. In: Proc. of ICSE 2005 (2005)
9. Galeotti, J.P., Rosner, N., Lopez Pombo, C., Frias, M.: Analysis of Invariants for Efficient Bounded Verification. In: ISSTA 2010 (2010)
10. Galeotti, J.P., Rosner, N., Lopez Pombo, C., Frias, M.: TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. IEEE Trans. Soft. Eng. (2013)
11. Geldenhuys, J., Aguirre, N., Frias, M.F., Visser, W.: Bounded Lazy Initialization. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 229–243. Springer, Heidelberg (2013)
12. Iosif, R.: Symmetry Reduction Criteria for Software Model Checking. In: Bošnački, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 22–41. Springer, Heidelberg (2002)
13. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: Software Verification Platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)
14. Jackson, D.: Software Abstractions. MIT Press (2006)
15. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL 2010 (2010)
16. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated Verification of Shape and Size Properties via Separation Logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
17. Nguyen, H.H., Chin, W.-N.: Enhancing program verification with lemmas. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 355–369. Springer, Heidelberg (2008)
18. Parrino, B.C., Galeotti, J.P., Garbervetsky, D., Frias, M.F.: A Dataflow Analysis to Improve SAT-Based Bounded Program Verification. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 138–154. Springer, Heidelberg (2011)
19. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proceedings of LICS 2002 (2002)
20. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
21. Visser, W., Pasareanu, C.S., Pelanek, R.: Test Input Generation for Java Containers using State Matching. In: ISSTA 2006 (2006)
22. Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using Boolean satisfiability. ACM TOPLAS 29(3) (2007)