Computing Exact Loop Bounds for Bounded Program Verification

Tianhai Liu[1], Shmuel Tyszberowicz[2], Bernhard Beckert[1], and Mana Taghdiri[3]

[1] Karlsruhe Institute of Technology, Germany
[2] RISE, Southwest University, China and The Academic College Tel Aviv Yaffo, Israel
[3] Horus software GmbH, Germany

Abstract. Bounded program verification techniques verify functional properties of programs by analyzing the program for user-provided bounds on the number of objects and loop iterations. Whereas those two kinds of bounds are related, existing bounded program verification tools treat them as independent parameters and require the user to provide them. We present a new approach for automatically calculating *exact* loop bounds, i.e., the greatest lower bound and the least upper bound, based on the number of objects. This ensures that the verification is complete with respect to all the configurations of objects on the heap and thus enhances the confidence in the correctness of the analyzed program. We compute the loop bounds by encoding the program and its specification as a logical formula, and solve it using an SMT solver. We performed experiments to evaluate the precision of our approach in loop bounds computation.

1 Introduction

Bounded program verification techniques (e.g. [7, 15, 24]) verify functional properties of object-oriented programs, where loops are unrolled and the number of objects for each class is bounded. These techniques typically encode the program and the property of interest into a logical formula and check the satisfiability of the formula by invoking an SMT solver. They provide an attractive trade-off between automation and completeness. They automatically exhaustively analyze a program based on the user-provided bounds [4] and thus guarantee to find any bug (with respect to the analyzed property) within the bounds, but defects outside bounds may be missed. As a result, bounded program verification has becomes an increasingly attractive choice for gaining confidence in the correctness of software.

Existing bounded program verification techniques typically require the user to provide two kinds of bounds as separate parameters: (1) the *loop bounds* that limit the number of iterations for each loop, and (2) the *class bounds* that limit the number of objects for each class. (The class bound for a primitive type, e.g., the Java `int`, is the size of integer bit-width.) These two kinds of bounds,

[4] They analyze a program based on *both* bounds—objects and loop iterations. Thus, not all object space within bounds is necessarily explored (as explain in what follows).

however, are not independent and have to be chosen carefully; the class bounds can affect the number of loop iterations, and the loop bounds can influence the size of object space to be explored. To clarify, consider the following loop that traverses an acyclic singly-linked list in Java, starting from the header entry `l.head`: `e=l.head; while(e!=null){ e=e.next;}`.
Supposing the loop bound is 2, we unroll the loop twice and then add an `assume` clause which fails if it iterates more than 2 times. The code will be:
`e=l.head; if(e!=null){ e=e.next; if(e!=null){ e=e.next;assume(e==null);}}`.
When the list has at most one element (implied, e.g., by the provided class bound), the second `if`-condition always evaluates to `false` and thus the following code is unreachable. On the other hand, if the list contains at least 5 elements (implied, e.g., by the specification or by the rest of the code), the encoding formula of the code evaluates to `false`, since the `assume` statement will never evaluate to `true`. Furthermore, when the list contains up to 5 elements, only the lists with at most 2 elements are analyzed and the other elements in the list are not treated.

When a loop is unrolled too many times (i.e., more than the least upper bound on the number of loop iterations), the unrolled program has many unreachable paths which may impede the performance of the underlying solver. The verification process may fail due to the solver being overloaded. When a loop is unrolled too few times (i.e., fewer than the greatest lower bound on the number of loop iterations), none of the program executions that reach the loop will be valid in the unrolled program, thus any property concerning the loop will vacuously hold in all runs. This is also the case for *infinite* loops; we consider a loop infinite when it does not terminate for any input. Selecting a loop bound that lies strictly between the greatest lower- and the least upper-bound causes the analysis to be incomplete (i.e., it explores only a part of object space).

Several approaches have been developed to compute loop *upper* bounds (e.g. [4, 10, 18, 22]), and many of them (e.g. [4, 10, 18]) do not require a specific bound for objects; they compute loop upper bounds as functions, based on the input sizes. However, none of those approaches can handle arbitrary configuration of objects on the heap. They either focus only on primitive types [4, 18, 22] or support only particular configuration of objects [10]. Furthermore, none of those approaches considers specifications in computing loop bounds, and many of them compute a valid upper bound, which is not necessarily the *least* upper bound.

Incremental bounded model checkers, e.g. NBIS [11], also can be used to compute loop upper bounds. Starting from an initial number, they unroll a loop and check whether the loop condition still holds after the last unrolled iteration. If so, a new upper bound candidate is found and checked again iteratively. This approach, however, is imprecise in the presence of class bounds and specifications. It may compute upper bounds that are higher than the least upper bound, thus many unreachable paths arise in the unrolled program, and the verification may fail. To overcome this potential failure, the user may restart verification with smaller class bounds. Thus the confidence in the correctness of the code is reduced, as the number of relevant objects is smaller. Moreover, this approach does not compute bounds when the loops are *non-terminating*. A loop is considered

non-terminating when it does not terminate for at least one input while it may terminate for other inputs. This is inconsistent with bounded program verification tool, which do analyze the terminating executions of a method and ignore non-terminating runs.

We present an approach that is meant to be used as a pre-processing phase in bounded program verification. It focuses on data-structure-rich programs and can handle arbitrary configurations for the objects in the heap. Given both a program method m selected for analysis and class bounds b, we compute both the *greatest* lower bound and the *least* upper bound for each loop that is reachable from m. Our approach, therefore, can provide the user with an insight on what loop bounds to consider in bounded program verification, and to enhance the confidence for program correctness with respect to the class bounds. When a method specification exists, we consider it as well. In addition to numerical bounds, we also output a pre-state (and an execution trace) that witnesses each computed bound, which guarantees that the computed bounds are feasible. We produce loop bounds even for a non-terminating loop, provided that the loop has at least one terminating execution; recall our definition of non-terminating. This is consistent with bounded program verification approaches. Besides, we can detect unreachable loops by analyzing the results of bound computation.

We compute loop lower- and upper-bounds for Java programs annotated in a subset of JML (Java Modeling Language) [13]. We translate the code, its precondition, and its sub-routines' specifications (if they exist) into a first-order formula, encoding the loops' effects as recursive uninterpreted functions. The resulting formula is solved for the *exact* greatest lower and least upper bounds for each loop. This is achieved by calling an SMT (satisfiability modulo theories) solver that is able to solve optimization problems. Given a formula f and an optimization objective o, the SMT solver finds a model for f to achieve the goal o. Several off-the-shelf SMT solvers have been extended to solve optimization problems, e.g., Z3 [19] with νZ [3] and SYMBA [14], MathSAT5 [5] with OptiMathSAT [21]. Besides, some SMT-based algorithms for solving optimization problems have been developed, e.g., the authors of [17] integrated an SMT solver with a classical incremental solving algorithm to solve generic optimization problems, and an algorithm in [20] aims to solve linear arithmetic problems. Our approach takes advantage of these recent advances in SMT solvers. Our target logic is undecidable, i.e., it is possible for the underlying solver to output '*unknown*'. However, for the small class bounds generally used in bounded program verification, the solver returns a definite answer.

We have implemented our approach and NBIS' approach in prototype tools *BoundJ* and *IncUnroll*, respectively. We compared the computed bounds using these tools. Our experiments reveal that in all cases *BoundJ* has computed precise loop bounds, while *IncUnroll* does not. On the other hand, *IncUnroll* can produces loop bounds with the increased class bounds, while *BoundJ* returns '*unknown*' for large class bounds.

2 Logical Formalism

We focus on analyzing object-oriented programs, and currently support a basic subset of Java—excluding floating-point numbers, strings, generics, and concurrency. We support class hierarchy without interfaces and abstract classes. Any method which is called by the analyzed method and has no specification is inlined into its call sites; otherwise it is replaced by its specification. Specifications are written in JML [13], and should not include exceptional behaviors and model fields. The constructs `requires` and `ensures` define, respectively, a method's pre- and postcondition. We support arbitrarily nested universal and existential quantifiers, and allow using the JML reachability construct.

We translate the given code and its specifications into a first-order SMT logic that consists of quantified bit-vectors, unbounded integers, and uninterpreted functions. We now describe our target logic. For this we use the SMT-LIB 2.0 syntax [2], in which expressions are given in a prefix notation. The command `(declare-fun f (A1 .. An-1) An)` declares a function $f : A_1 \times .. \times A_{n-1} \rightarrow A_n$. Constants are functions that take no arguments. The command `(assert F)` asserts a formula F in the current logical context; multiple assert statements are assumed to be implicitly conjoined. The `(push)` command pushes an empty assertion set onto the assertion-set stack. and `(pop)` pops the top assertion set from the stack. The command (`check-sat`) triggers solving a conjunction of formulas. The operator `ite` denotes a ternary if-then-else expression. Basic formulas are combined using the boolean operators `and`, `or`, `not`, and `=>` (implies). Universal and existential quantifiers are denoted by the keywords `forall` and `exists`. The Z3 solver contains two extensions of SMT-LIB to express optimization objectives. The command (`maximize t`) instructs the solver to produce a model that maximizes the value of the integer term `t` and to return the assignment for `t` in the solution, if one exists. The (`minimize t`) command finds the smallest value of `t`.

Our translation uses the fixed-size bit-vectors theory, in which sorts are of the form `(_BitVec m)`, where m is a non-negative integer denoting the size of the bit-vector. Bit-vectors of different sizes represent different sorts in SMT. This theory models the precise semantics of unsigned and of signed two-complements arithmetic, supporting a large number of logical and arithmetic operations on bit vectors. The translation also uses the unbounded integers theory, which contains only one sort `Int`, corresponding to integer numbers. It supports arithmetic operations to be applied to numerical constants or variables.

3 Motivating Example

We use two examples to illustrate that we compute precise loop bounds. Figure 1 shows a Java program for a check-in process in a youth hostel. The check-in process requires that the guests will be young (the precondition of the method `checkin`), and the group size is neither greater than 10 nor less than 3 (the postcondition[5] of the method `openRoomFor`). Carefully inspecting the code, we

[5] We intentionally refer to the postcondition rather than to a precondition that limits the number of guests in order to demonstrate our approach.

```
1 int guest = 0;
2 //@requires (\forall int i; 0<=i&&i<ages.length; 0<ages[i]&&ages[i]<=18);
3 void checkin(int[] ages) {
4     for (int i = 0; i < ages.length; i++) {
5       if (ages[i] <= 27) guest++;}
6     openRoomFor();}
7 //@ ensures 3<=\old(guest) && \old(guest)<=10;
8 native void openRoomFor();
```

Fig. 1: A simple program for youth hostel checkin. The method `openRoomFor`'s postcondition at line 7 constrains the range of values of `guest`.

notice that the branch condition at line 5 is implied `true` by the precondition of the method `checkin`. Thus the number of loop (line 4) iterations equals to the number of young guests (the field `guest` when the loop terminates). Suppose the Java `int` bit-width is 6 (i.e., integer numbers ranging from -32 to 31). We evaluate the loop lower- and upper bounds to 3 and 10, respectively. However, using an approach that computes loop bounds by incrementally unrolling loops (Section 1), the loop upper bound is 31, since it does not consider the postcondition of the method `openRoomFor` when evaluating the loop condition. The incremental loop unrolling approach provides no lower bounds computation.

Figure 2 illustrates a Java implementation of a `copy` method for a singly-linked list of `Data` entries. Given an instance `d` of `Data`, the `copy` method deeply copies the receiver list, starting from the first occurrence (exclusive) of `d`. If `d` does not exist, nothing is copied. Bounded program verification techniques analyze programs with respect to specific bounds on the number of objects. Assume that the maximum number of objects of type `List`, `Entry`, and `Data` is 2, 26, and 1, respectively. The bounds for each loop are computed separately. For instance, when the upper bound of the second loop (*Loop2* at line 9) is computed, no specific bound for the preceding loop (*Loop1* at line 4) is assumed. Our technique computes the loop upper bounds considering all the cases in which the loops terminate. Loop1 may not terminate for some inputs, e.g., when the receiver list is a cyclic one. For all inputs for which Loop1 terminates our technique outputs 26 as the upper bound and generates a witness in which an acyclic list contains 26 entries, where the last one is followed by `null`. For all inputs for which Loop2 terminates, our technique outputs 12 as Loop2's upper bound and generates a witness where an acyclic list has 13 entries, where the first one has `Data d`. That makes sense, because the `copy` method deep copies an acyclic linked list, and fresh entries one-to-one correspond (excluding the entry containing d) to the entries in the receiver list. Thus Loop2 can allocate at most 12 fresh objects, and a total of 25 ($= 12 * 2 + 1$) `Entry` objects are used. If Loop2 iterates 13 times, e.g., a total 27 objects are needed, which is larger than the bound (26) on `Entry`.

As shown, the number of loop iterations heavily depends on both the specifications and the number of objects in the analyzed domain. Furthermore, the number of iterations of different loops is inter-dependent. Detecting those dependencies manually and computing the precise loop bounds can be prohibitively difficult,

```
1  class List { Entry head;
2    List copy(Data d) {
3      Entry curr = head;
4      while (curr != null && curr.data != d) curr = curr.next; // Loop 1
5      List result = new List();
6      if (curr != null) {
7        curr = curr.next;
8        Entry last = null;
9        while (curr != null) {   // Loop 2
10         Entry e = new Entry(curr.data);
11         if (last == null) { result.head = e; }
12         else { last.next = e; }
13         last = e;
14         curr = curr.next;
15       }
16     }
17     return result;}}
18 class Entry {/*@ nullable */ Entry next; /*@ nullable */ Data data;
19     Entry(Data d) { data = d; next = null;}}
20 class Data {}
```

Fig. 2: A Java program to perform a deep copy on a linked list.

thus an automatic approach for computing exact loop bounds (in the presence of specifications and class bounds) can significantly enhance the bounded program verification engineers' confidence in the correctness of the analyzed programs.

4 Our Approach

Given a method p selected for analysis from a piece of code and a set of class bounds b, our technique computes for each loop l two numbers, GLB_l and LUB_l. They respectively denote the *greatest lower bound* and the *least upper bound* on the number of iterations of l, since we ensure that no valid execution of p can iterate the loop l less times than GLB_l or more times than LUB_l. In order to analyze only valid executions, we consider the whole code when computing the bounds for a loop l. For each computed bound the output also contains a witnessing pre-state and an execution trace.

We translates the given method p and its constraints (specifications) c (consisting of p's precondition and the annotations of the methods reachable from p) into an SMT formula, based on a set of user-provided bounds b on the number of instances of the classes. Let T denote this translation; then $T[p, c, b]$ produces a tuple (s, f, N_l), where s is the pre-state of p, f is an SMT formula that encodes the control flow and dataflow of p and the additional constraints b and c, and N_l represents the number of times the loop condition has been checked for loop l. We distinguish various loops inside p using loop ids.

The LUB_l is computed by delegating a formula of the form $f \wedge exit(l, N_l) \wedge maximize(N_l)$ to an SMT solver that provides the functions to solve optimization

problems. The function *exit* means that the loop l exits after checking the loop condition N_l times. The *maximize* command instructs the solver to find a model where N_l is the biggest compared to the values in other models. A satisfying solution to this formula represents a terminating execution of p in which the loop l is reachable when running p and the number of iterations of l is $N_l - 1$ ($N_l > 0$), and unreachable in case that $N_l = 0$. When the formula is unsatisfiable, it means that either the user-provided class bounds are too small or the methods are over-specified (e.g., the precondition of the analyzed method is `false`). The GLB_l is computed similarly to least upper bound computation, using the command *minimize* instead of *maximize*.

Unbounded integers are used in our translation to encode loop iterations. Since (1) a loop may not terminate, and (2) even for terminating loops, the number of iterations is not known and thus cannot be bounded apriori. Hence, our target logic is undecidable, and it is possible for the solver to output 'unknown'. In such a case, our analysis terminates with no conclusive outcome.

4.1 Encoding Control Flow

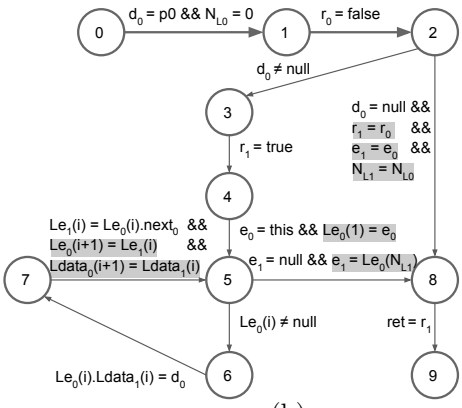
We encode the control flow of the analyzed method using a *computation graph* [24]. Each node in this graph represents a control point in the program, and each edge represents either a program statement or a branch condition. There are exactly one node to entry the graph and one node to exit from the graph. If a loop in the analyzed method is triggered multiple times, due to either method invocations or that it is an inner loop, then multiple occurrences of the loop exist in the computation graph. We compute the loop bounds for each loop occurrence. This is consistent with bounded program verification.

Bounded program verification tools such as Jalloy [24] and InspectJ [15] also use computation graphs to encode control flow. However, they unroll loops and thus assume that the graph is acyclic. In that case, control flow can be encoded by simple boolean variables. Our approach, on the other hand, preserves loops as cycles in the graph and encodes their (cyclic) control flow via uninterpreted functions in the SMT logic. More precisely, similarly to previous approaches, we encode an edge that does not belong to any loop from node m to node n, using a boolean variable E_{mn}, whose truth value denotes whether the edge is traversed or not. When an edge belongs to a loop, the encoding must clarify in which loop iterations the edge is traversed. Therefore, when an edge from m to n belongs to loop l, we encode it using a boolean-valued, uninterpreted function $E_{mn} : Int^{>0} \rightarrow Bool$ ($Int^{>0}$ denotes positive integers). The expression $E_{mn}(i)$ evaluates to true if the edge is traversed in the i^{th} iteration of l. The exit edge of l is traversed only once the loop condition is not fulfilled for the $(N_l)^{th}$ iteration of the loop. We encode the exit edge of a loop l as the expression $E_{mn}(N_l)$, where $N_l > 0$ and $N_l = 1 + K$, where K is the number of iterations of the loop l.

We use the term *entry edge* (*exit edge*) to denote an edge that leads to the entry node (exits from the exit node) of a loop but does not belong to that loop. We use the term *head edge* (*tail edge*) to denote the first (the last) edge of a loop. The control flow for the computation of the loop bounds is encoded using the

```
class Entry { Entry next;
  Data data;
  boolean assign(Data d){
    boolean r = false;
    if(d != null){
      r = true;
      Entry e = this;
      while(e != null){
        e.data = d;
        e = e.next;
      }
    }
    return r;}}
class Data {}
```

(a)



(b)

```
(1)   (assert E_0_1)
(2)   (assert (=> E_0_1 E_1_2))
(2)   (assert (=> E_1_2 (or E_2_3 E_2_8)))
(2)   (assert (=> E_2_3 E_3_4))
(2)   (assert (=> E_3_4 E_4_5))
(2)   (assert (=> E_2_8 E_8_9))
(2,3)(assert(=> E_4_5(or(E_5_6 1)(and(E_5_8 N_L_1)(= N_L_1 1)))))
(2)   (assert (forall ((i Int)) (=> (E_5_6 i) (E_6_7 i))))
(2)   (assert (forall ((i Int)) (=> (E_6_7 i) (E_7_5 i))))
(2,4)(assert (forall ((i Int)) (=> (E_7_5 i) (or (E_5_6 (+ i 1))
                        (and (E_5_8 N_L_1) (= N_L_1 (+ i 1)))))))
(2)   (assert (=> (E_5_8 N_L_1) E_8_9))
```

(c)

Fig. 3: (a) sample Java code, (b) its computation graph, (c) our control flow encoding. The character 'L' in (b) denotes loop id. The variable N_{L0} represents the number of times that the loop condition has been checked. After exiting the loop it is renamed to N_{L1}, and is encoded in (c) as the SMT variable `N_L_1`.

following four general rules. (1) The first edge of the computation graph must be traversed. (2) If an edge E_{mn} is traversed, at least one of the outgoing edges of node n must be traversed (dataflow constraints prevent more than one outgoing edge from being traversed). If node n belongs to a loop l, the iteration index must be considered. In particular, if n is the loop's head node, then (3) if a head edge is traversed, either the first iteration starts or the loop exits before the first iteration, and (4) if a tail edge at the i^{th} iteration of the loop is traversed, then either the $(i+1)^{th}$ iteration starts or the loop exits before this iteration.

Figure 3 provides an example. Figure 3(a) shows a Java method that sets the field `data` of all list elements to the input value, provided that this value is not `null`. Figure 3(b) gives the corresponding computation graph. The edge labels denote the statements and branch conditions in a special SSA-like format as described in Section 4.2. Figure 3(c) presents our encoding of the control flow. In this example, `E_0_1`, `E_1_2`, `E_2_3`, `E_2_8`, `E_3_4`, `E_4_5`, and `E_8_9` are boolean variables, while `E_5_6`, `E_6_7`, `E_7_5`, and `E_5_8` are boolean-valued

```
(assert (=> E_0_1 (and (= d_0 p0) (= N_L_0 0))))
(assert (=> E_1_2 (= r_0 false)))
(assert (=> E_2_3 (not (= d_0 null_Data))))
(assert (=> E_3_4 (= r_1 true)))
(assert (=> E_4_5 (= e_0 this)))
(assert (=> E_2_8 (= d_0 null_Data)))
(assert (=> E_8_9 (= ret r_1)))
(assert (=> (E_5_8 N_L_1) (= e_1 null_Entry)))
(assert(forall((i Int))(=>(E_5_6 i)(not(=(L_e_0 i)null_Entry)))))
(assert(forall((i Int))(=>(E_6_7 i)(forall((e Entry))
      (=(L_data_1 i e)(ite(= e (L_e_0 i))d_0(L_data_0 i e)))))))
(assert (forall ((i Int)) (=> (E_7_5 i)
                              (= (L_e_1 i) (next_0 (L_e_0 i))))))
```

(a)

```
(assert(=> E_4_5 (and (= (L_e_0 1) e_0)
                      (= (L_data_0 1) e_0) (data_0 e_0))))
(assert (=> (E_5_8 N_L_1) (= (L_e_0 N_L_1) e_1)))
(assert(forall((i Int))(=>(E_7_5 i)(and(=(L_e_0(+ i 1))(L_e_1 i))
      (forall((e Entry))(=(L_data_0(+ i 1) e)(L_data_1 i e)))))))
(assert (=> E_2_8 (and (= r_1 r_0) (= e_1 e_0) (= N_L_1 N_L_0))))
```

(b)

Fig. 4: Dataflow Encoding SMT formulas: (a) dataflow (b) frame conditions. Each class has a distinct null value, e.g., the `null_Data` and the `null_Entry`.

functions. The edge `E_4_5` is an entry edge, `E_5_6` is a head edge, `E_7_5` is a tail edge, and `E_5_8` is an exit edge. The numbers preceding the constraints correspond to the four encoding rules presented above. For each loop l in the computation graph, we introduce an integer variable N_l to represent the number of times that the loop condition has been checked; e.g., N_{L0} and N_{L1} in Fig. 3(b) and `N_L_1` (that encodes N_{L1}) in Fig. 3(c).

4.2 Encoding Dataflow

We now provide an overview of our encoding of Java statements, which is based on the InspectJ approach [15]. We focus on how loops affect the encoding. The types that are accessed in the analyzed code are encoded using bit-vectors in the SMT logic. That is, if a Java type T is bounded by the user-provided number n, we encode T as a bit-vector of size $\lceil log(n+1) \rceil$ (including the *null* value). In the following description, we use $BV[T]$ to represent the bit-vector of a Java type T.

In an acyclic computation graph, all variables and fields of the program are renamed so that they are assigned at most once along each path of the graph. Since our computation graphs can be cyclic, renaming cannot be achieved by enumerating all paths. We rename variables and fields of the program assuming that each loop constructs a separate naming context (similar to a called method, e.g.). This separates the naming of variables (fields) in one loop from the others, which makes it easier to support complex loop structures. More precisely, renaming variables (fields) involves the following steps. (1) Starting from the innermost

loop l, we give any variable (field) that may be updated by l an initial name, and then perform renaming within the body of l as for an acyclic computation graph. (2) We collapse the cycle (loop) l of the computation graph into a single node, denoting the initial and the final names of the variables updated in l. (3) We repeat step 1, considering the collapsed loops. Hence, any time a collapsing node is visited, adequate conditions are produced to ensure that the variables (fields) of the current context hold the same values as the initial/final variables (fields) of the collapsed loop. In the example in Fig. 3(b), d_0, N_{L0}, N_{L1}, e_0, e_1, r_0, r_1, and $data_0$ belong to the outer context, whereas Le_0, Le_1, $Ldata_0$, and $Ldata_1$ belong to the loop context. Since the loop does not update the $next_0$ field and the constants, e.g., $this$, $p0$ and ret, both contexts share them. Data accesses outside loops are encoded as follows: A variable v of type T is encoded as an SMT variable $v : BV[T]$, and a field f of type T_2 declared in a class T_1 is encoded as a function $f : BV[T_1] \rightarrow BV[T_2]$. However, if a variable or field is updated within a loop, one needs to know the updates performed in each loop iteration. A variable v_l of type T that may be modified within a loop l is encoded as a function $v_l : Int^{>0} \rightarrow BV[T]$, where $v_l(i)$ denotes the value of v in the i^{th} iteration of l. Similarly, a field f of type T_2 declared in a class of type T_1, that may be modified within a loop l, is encoded as a function $f_l : Int^{>0} \times BV[T_1] \rightarrow BV[T_2]$, $f_l(i, o)$ denotes the value of $o.f$ in the i^{th} iteration of the loop. Figure 4(a) shows the dataflow formulas for Fig. 3(b). The first 8 formulas correspond to the edges outside the loop. The last 3 encode the dataflow in each loop iteration.

Frame Conditions. Frame conditions are used to avoid underspecification of nodes with multiple incoming edges, and to ensure the correctness of the dataflow. The highlighted expressions in Fig. 3(b) are the frame conditions for the corresponding dataflow expressions. Lets take the merge node 8 for example. Since the variables r, e, and N_L are updated (and thus renamed) only in the path $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 8$, frame conditions for these variables are required when the path $2 \rightarrow 8$ is taken; the last formula in Fig. 4(b) is the relevant frame condition. Special frame conditions are required when the merge node is a loop's head node, e.g., node 5 in Fig. 3(b). Before the first iteration, $Le_0(1)$ equals e_0 and $Ldata_0(1)$ equals $data_0$ (encoded as the first formula in Fig. 4(b)). After the last iteration, e_1 equals $Le_0(N_l)$ (encoded as the second formula in Fig. 4(b)). Furthermore, in each new iteration $i + 1$, $Le_0(i + 1)$ equals $Le_1(i)$ and $Ldata_0(i + 1)$ equals $Ldata_1(i)$ (encoded as the third formula in Fig. 4(b)). It should be noted that the frame condition for N_l ensures that the variable `N_L_1` equals to the number of times that the loop condition has been checked. Without the frame condition we may get wrong loop bounds, due to traversing spurious paths.

Nested Loops. If a loop l_2 is nested in a loop l_1, the iterations of l_2 depend on the iterations of l_1. Therefore, we encode those variables (fields) that are updated in the inner loop l_2 by adding an additional iteration column to the SMT functions that represent those variables (fields). That is, if a variable v^{l_2} of type T is modified within l_2, we declare an SMT function $v^{l_2} : Int^{>0} \times Int^{>0} \rightarrow BV[T]$, where $v^{l_2}(i_1, i_2)$ denotes the value of v^{l_2} in the $(i_2)^{th}$ iteration of l_2 while in the $(i_1)^{th}$ iteration of l_1. Updated fields are encoded in a similar way by adding an

additional column. Moreover, the edge variables encoding the control flow of l_2 will also get an additional column corresponding to the iteration number of the outer loop. It works the same way for any depth of nesting.

Encoding Specifications. Computing loop bounds does not require any user-provided specifications or annotations; the user only provides bounds on the number of elements of each type. However, if the precondition of the analyzed method or the method contracts of the called methods are provided, our analysis will take them into account. That is, if the user provides method contracts for an invoked method, they will be used to substitute any call to the method. Otherwise, the method body will be inlined in its call sites. More details can be found in our previous work [15].

4.3 Computing Loop Bounds

In order to compute the bounds of a loop l, we constrain l to terminate, that is, its exit edge to be traversed when its loop condition has been checked N_l times, if l is reachable from the analyzed method. We also trigger the solver to find the model where N_l has the maximal assignment. We solve the conjunction of all the control flow, dataflow, frame conditions, and specification formulas. If this formula is satisfiable, N_l is assigned a value in the satisfying solution, where $N_l > 0$ denotes the loop l is reachable and its bound is N_l, $N_l = 0$ denotes l is not reachable. If the formula is unsatisfiable, then either the user-provided class bounds are not large enough or the user-provided specifications are not consistent by themselves. The following formulas give the SMT commands that computes the least upper bound for the example of Fig. 3.

```
(push) (assert (=> (> N_L_1 0) (E_5_8 N_L_1)))
(maximize N_L_1) (check-sat) (get-model) (pop)
```

To compute a loop lower bound, we just replace the *maximize* command by *minimize*. It is possible for the solver to output 'unknown' because our logic is undecidable, and then our analysis terminates with no conclusive outcome.

5 Experiments

Our prototype tool (*BoundJ*) uses: the Jimple 3-address intermediate representation provided by the Soot optimization framework [23] to preprocess Java program code; the Common JML Tools package (ISU) [13] to preprocess JML specifications; and Z3 version 4.4.2 [19] as the underlying SMT solver. We also have considered the approach used in NBIS [11]to evaluate the precision of our approach. Since NBIS targets C and C++ code, and does not consider specifications or class bounds, we implemented that approach in a prototype tool (*IncUnroll*) that targets Java and accepts the same inputs that *BoundJ* does. We report on a collection of benchmarks, selected from InspectJ [15] (a bounded program verification system), KeY [1] (a program verification system), JDK (Java Development Kit), and TPDB (Termination Programs Data Base) [25]. All the experiments[6] have been performed on an Intel Core 2.50 GHz with 4 GB of RAM using Linux 64bit.

[6] The complete benchmarks can be found at `http://asa.iti.kit.edu/478.php`.

Method	Scope Size	GLB (*BoundJ*)	LUB (*BoundJ*)	LUB (*IncUnroll*)
	3	X	X	3
BinaryHeap	4	1	1	4
deleteMin	5	1	1	5
	6	1	2	6
	4	5, 0	5, 0	7, 1
KeYList.	5	5, 0	6, 0	15, 2
removeDup	6	5, 0	7, 1	31, 3
	7	5, 0	8, 2	63, 4
	8	5, 0	9, 3	127, 5
	3	0, 0	3, 1	?, ?
OurList.	4	0, 0	4, 1	?, ?
copy	5	0, 0	5, 2	?, ?
	6	0, 0	6, 2	?, ?
	3	1, 1	1, 2	1, 2
JDKList.	4	1, 1	3, 4	3, 4
add	5	1, 1	7, 8	7, 8
	7	1, 1	21, ?	21, 32
	10	1, 1	?, ?	175, 256
	3	0	9	?
NonTerm.	4	0	10	?
fibonacci	5	0	10	?
	6	0	10	?
	6	0	63	63
NonTerm.	7	0	?	127
gause	9	0	?	508

Table 1: Results of Computing Loop Bounds using *BoundJ* and *IncUnroll*.

The evaluation results are shown in Table 1. The *Method* column shows the names of the entry methods of the analyzed programs. There are in total 6 programs have been analyzed. To increase the complexity of the specifications, we also added special method contracts for the sub-routines (if exist). The method *deleteMin* of the class *BinaryHeap* (that calls another 3 methods, 109 LOC) effectively extracts the minimum element in a *min heap*[7] and restores the properties of min heap. The *removeDups* method of the class *KeYList* (3 methods, 33 LOC) removes the duplicate elements from a queue. The *add* method (2 methods, 39 LOC) is classical implementation in JDK 1.7. All those methods have complex preconditions, i.e., quantifiers have been involved. The methods *deleteMin* and *add* also use JML reachability expressions to constrain the heap configurations. The *copy* method is the method presented in Section 3. In addition to these data-structure-based benchmarks, we have also used benchmarks that involve only primitive types. Such benchmarks are typical for the loop bound computation and the non-termination detection communities. They (the methods `fibnacci` and `gause`) do not have any specification and there are around 10 LOC in average in each method, however, we have selected these benchmarks for

[7] A min heap is a binary heap where the values that are stored in the children nodes are greater than the value stored in the parent node.

the following two reasons: (i) The number of iterations for many of these loops is non-linearly distributed. Therefore, computing their loop bounds is particularly challenging in many existing approaches. (ii) To validate that our approach indeed computes loop bounds for the methods that contain at least one terminating path. For each analyzed program, we have exploited each tool to compute ∼4 loop upper bounds for different *class bounds*, and in total 25 loop upper bounds computations have been done using each tool. Besides, we also used *BoundJ* to compute the loop lower bounds. Thus in total we have done 75 computations.

The *Scope Size* column shows the bounds on the number of objects of each class and on the size of the integer bit-width. For a scope size n, the analyses of *deleteMin*, *removeDups*, and *copy* methods have to explore data spaces of size $(n+1)^{11} * 2^{9n}$, $(n+1)^9 * 2^n$, and $(n+1)^5 * 2^n$, respectively. (The numbers are calculated based on the number of accessed classes, fields, and parameters. Computations are skipped for space reasons.) The columns *GLB* and *LUB* represent the computed lower and upper bounds, respectively. When a method contains more than one loop, the bounds are shown as a sequence of numbers separated by commas. The symbol X denotes that the loop is not reachable from the method. Question marks (?) mean that no definite answer is achieved after the timeout limit of 20 minutes.

In Table 1 we observe that: (i) *BoundJ* computed *exact* loop lower-/upper-bounds for all data structure-rich methods. A careful inspection of the code reveals that all computed loop lower- and upper bounds are greatest and least, respectively. (ii) *IncUnroll* does not always compute precise loop bounds as *BoundJ* does. Since *IncUnroll* does not consider the whole code in loop bounds computation, on average its computed loop upper bounds are 4.2 times (median 4, maximum 13) greater than the ones *BoundJ* computed. In addition, *IncUnroll* failed to compute the loop upper bounds for the non-terminating methods `copy` and `fibonacci` because of timeout, while *BoundJ* still produced the loop upper bounds for the methods since it considers all program executions that terminate. (iii) Nevertheless, *IncUnroll* can compute loop upper bounds with increased class bounds, while *BoundJ* timeouts for 2 (of 25) cases. According to the *small scope hypothesis* [12], bounded program verification systems aim to analyze the program with respect to a small scope. Furthermore, *BoundJ* always produces the *exact* loop bounds based on the user-provided class bounds, thus it guarantees any bounded program verification is complete for the given class bounds and enhances the confidence in the correctness of the analyzed program.

6 Related Work

Various techniques have been developed to compute loop bounds in real-time systems. To estimate loop bounds, they either require annotations [8] or perform a numerical analysis to achieve a numerical interval of loop upper bounds [6]. To achieve better analysis performance and more precise loop bounds, the technique described in [16] employs a combination of abstract interpretation, inter-procedural program slicing, and inter-procedural dataflow analysis. Unlike our technique that requires explicit user-provided class bounds, the techniques de-

scribed in [4, 9, 22] generate symbolic bounds (functions) in terms of loop inputs. All these approaches, however, focus on numerical loops; some of them (e.g. [4, 6]) even require well-structured loops with no branches inside them. Our approach can work on arbitrary loops and target data structure-rich programs.

To compute bounds for complex loops in C++ code, SPEED [10] generates computational complexity bound functions that contain well-implemented abstract data structures. For loops that access data structures, it generates symbolic bound expressions in terms of numerical properties of the data structures and user-defined quantitative functions. Generating symbolic bounds depends on the generation of loop invariants. SPEED, however, requires user-provided specifications, does not support complex heap configurations (e.g., transitive reachability) in the precondition of the analyzed method, and in some cases outputs only an approximate loop bound functions.

The incremental bounded model checker NBIS [11] can be used to compute loop upper bounds. It instruments every loop in a given C/C++ code with an unrolling assertion that checks whether the loop can iterate beyond the current loop bound. The encoding of the code along with the unrolling assertions is checked for satisfiability. A satisfiable instance denotes an execution trace in which a loop iterates more times than its current bound. That loop is then unrolled according to the newly-found bound and the process starts over. However, such an approach has the following three drawbacks. (1) It does not terminate in case of non-terminating loops. (2) It does not always return the least upper bound, because the trace corresponding to a satisfying instance stops the execution when exiting the loop with a new bound, hence the code following the loop is ignored. [8] Preventing the trace from stopping is not possible, since it requires an encoding that does not depend on the loops being unrolled (since the needed number of unrolling is unknown prior to invoking the satisfiability procedure),[9] (3) This approach reduces the necessary confidence in the correctness of the analyzed code. The over-approximated loop upper bounds result in many unreachable paths after loop unrolling. Hence, the formulas that are translated from the unreachable paths may overload the underlying solver and cause the verification process to fail. It might be due to the computed upper bound of the loop is too high or the user-provided class bounds are too big. When the verification engineer uses smaller class bounds than those in the previous run to re-compute the loop upper bounds, the new loop upper bounds still may be too high, since the computation ignores the code and specifications following the loop under consideration. If the

[8] For instance, when each iteration of the loop allocates one instance of class `A` and the code after the loop allocates 2 objects of type `A`. If $bound(A) = 5$, no valid execution of the code (with respect to class bounds) can iterate the loop more than 3 times, whereas the computed upper bound will be 5 when ignoring the code after the loop.

[9] An alternative checking whether the trace is valid with respect to the class bounds by executing (symbolically or dynamically) the whole code. Invalidity of the current instance, however, does not necessarily mean that the newly-found loop bound is impossible; it may still be that another satisfying instance can be valid and gives a higher loop bound. Thus, in the worst case, such a validity check requires enumerating all possible satisfying instances, which makes the approach impractical.

engineer arbitrarily selects smaller loop upper bounds, not all inputs concerning the class bounds are completely analyzed, thus the correctness of the code is not guaranteed for the class bounds. Consequently, although this iterative approach is successful for terminating loops in the absence of class bounds and specifications, it is not applicable in the context of bounded program verification since the class bounds are necessary and the confidence in the correctness of the code is not guaranteed for the class bounds.

7 Conclusion

We have presented an approach for computing *exact* loop bounds of a given loop for bounded inputs. Such an analysis is particularly useful for bounded program verification in which the user has to provide bounds on both the size of objects and the number of loop unrollings. Our approach provides the user with insight on what loop bounds to consider and enhances the confidence in the correctness of the analyzed programs. We focus on data-structure-rich programs and support arbitrary configurations of the objects on a heap. We translate the Java code and its JML specifications (excluding the postconditions of the entry method) into an SMT formula and solve it using an SMT solver that can solve optimization problems. We compared our approach with another one that incrementally unrolls a loop and checks whether the loop condition still holds after the last unrolled iteration. Experiments show that our method indeed produces the exact bounds, whereas the other method computes lower and upper bounds, which not necessarily are the exact ones. Our approach can assist the bounded program verification engineers to obtain complete confidence in the verification process. Although our analysis is not guaranteed to produce a definite outcome (due to the undecidability of the target logic), our experiments show that in practice the unknown outcome occurs for higher input bounds and not for the small bounds that are typically used in bounded program verification.

Bounded verification techniques unroll not only loops but also recursive methods. Currently we handle loops only; computing bounds for recursion is left for future work. To improve scalability, we will optimize the encoding formulas using quantify elimination techniques, e.g., symmetry breaking and pattern matching. We will also study the application of our approach to other areas, e.g., worst-case execution time and dynamic heap consumption program analysis.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book: From Theory to Practice, LNCS, vol. 10001. Springer (2016)
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.5. Tech. rep., The University of Iowa (2015)
3. Bjørner, N., Phan, A.D., Fleckenstein, L.: νZ - an optimizing smt solver. In: ETAPS. LNCS, vol. 9035, pp. 194–199. Springer (2015)

4. Blanc, R., Henzinger, T.A., Hottelier, T., Kovács, L.: ABC: algebraic bound computation for loops. In: LPAR. LNCS, vol. 6355, pp. 103–118. Springer (2010)
5. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS. LNCS, vol. 7795, pp. 93–107. Springer (2013)
6. Cullmann, C., Martin, F.: Data-flow based detection of loop bounds. In: WCET. OASICS, vol. 6. Schloss Dagstuhl (2007)
7. Dennis, G.D.: A Relational Framework for Bounded Program Verification. Ph.D. thesis, MIT (2009)
8. Gulavani, B.S., Gulwani, S.: A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In: CAV. LNCS, vol. 5123, pp. 370–384. Springer (2008)
9. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI. pp. 375–385. ACM (2009)
10. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: POPL. pp. 127–139. ACM (2009)
11. Günther, H., Weissenbacher, G.: Incremental bounded software model checking. In: SPIN. pp. 40–47. ACM (2014)
12. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT (2016)
13. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
14. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: POPL. pp. 607–618. ACM (2014)
15. Liu, T., Nagel, M., Taghdiri, M.: Bounded program verification using an SMT solver: A case study. In: ICST. pp. 101–110. IEEE (2012)
16. Lokuciejewski, P., Cordes, D., Falk, H., Marwedel, P.: A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: CGO. pp. 136–146. IEEE (2009)
17. Ma, F., Yan, J., Zhang, J.: Solving generalized optimization problems subject to SMT constraints. In: FAW-AAIM. LNCS, vol. 7285, pp. 247–258. Springer (2012)
18. Michiel, M.D., Bonenfant, A., Cassé, H., Sainrat, P.: Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In: RTCSA. pp. 161–166. IEEE (2008)
19. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
20. Sebastiani, R., Tomasi, S.: Optimization in SMT with $\mathcal{LA}$ ($\mathbb{Q}$) cost functions. In: IJCAR. LNCS, vol. 7364, pp. 484–498. Springer (2012)
21. Sebastiani, R., Trentin, P.: OptiMathSAT: A tool for optimization modulo theories. In: CAV. LNCS, vol. 9206, pp. 447–454. Springer (2015)
22. Shkaravska, O., Kersten, R., van Eekelen, M.: Test-based inference of polynomial loop-bound functions. In: PPPJ. pp. 99–108. ACM (2010)
23. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON. p. 13. IBM (1999)
24. Vaziri, M.: Finding Bugs in Software with a Constraint Solver. Ph.D. thesis, MIT (2004)
25. Termination problems data base (TPDB). `http://termination-portal.org/wiki/TPDB`, last accessed: June 2017.