Verified Resource Guarantees for Heap Manipulating Programs

Elvira Albert[2], Richard Bubel[1], Samir Genaim[2],
Reiner Hähnle[1] and Guillermo Román-Díez[3]

[1] CSE, Chalmers University of Technology, Sweden
[2] DSIC, Complutense University of Madrid (UCM), Spain
[3] DLSIIS, Technical University of Madrid (UPM), Spain

Abstract. Program properties that are automatically inferred by static analysis tools are generally not considered to be completely trustworthy, unless the tool implementation or the results are formally verified. Here we focus on the formal verification of *resource guarantees* inferred by automatic cost analysis. Resource guarantees ensure that programs run within the indicated amount of resources which may refer to memory consumption, to number of instructions executed, etc. In previous work we studied formal verification of inferred resource guarantees that depend only on integer data. In realistic programs, however, resource consumption is often bounded by the size of *heap-allocated* data structures. Bounding their size requires to perform a number of structural heap analyses. The contributions of this paper are (i) to identify what exactly needs to be verified to guarantee sound analysis of heap manipulating programs, (ii) to provide a suitable extension of the program logic used for verification to handle structural heap properties in the context of resource guarantees, and (iii) to improve the underlying theorem prover so that proof obligations can be automatically discharged.

1 Introduction

Formally proving the correctness of software can be crucial for many applications, e.g., in safety-critical systems. There are two possible approaches to certifying the correctness of software, (1) either perform full-blown verification of the correctness of the system or (2) alternatively validate its results for every execution. In the case of static analyzers, the first alternative is a daunting task, among other things, because of the sophisticated algorithms used for the analysis and their evolution over time. In this paper, we adopt the second alternative based on constructing a validating tool [14] which, after every run of the analyzer, formally (and automatically) confirms that the results are correct and, optionally, generates correctness proofs. Such proofs can then be translated to independently checkable *certificates* in the proof-carrying code style [6,13].

Resource usage analysis aims at (over-)approximating the amount of resources (time, memory, etc.) required to run a program in terms of its input arguments. COSTA [1,2] is a cost analyzer which allows the user to select a particular resource (among those available in the system) and automatically generate

resource usage upper bounds from Java bytecode (and hence Java) programs. Correctness of the techniques that COSTA implements is proven at the theoretical level, but the tool has not been formally verified. Thus, there is no guarantee that correctness is realized by the implementation. In recent work [3], we have proposed a fully automatic process of obtaining *verified resource guarantees* by using KeY [5], a state-of-the-art theorem prover for Java programs, for verifying that the upper bounds inferred by COSTA are correct. In essence, the COSTA and KeY systems cooperate in such a way that KeY produces formal correctness proofs for the different intermediate results used to obtain the upper bounds. When the resource guarantees depend only on data of integer type, this cooperation results in a fully automatic tool for producing verified resource guarantees.

However, it is often the case that resource guarantees depend on the structural properties of *dynamically allocated data*, e.g., the resource consumption of executing a loop that traverses a list is typically a function of the length of such a list. Resource analysis needs to keep track of how the size of data structures changes along the execution. For this purpose, COSTA integrates as an additional component the *path-length* analysis [17]. The path-length of a non-cyclic data structure is the length of the maximal path starting from the root, i.e., its depth. Inferring the path-length property also requires proving *acyclicity* of data structures and keeping track of possible *sharing* between pointers.

The main achievement of this paper is the extension of [3] to handle heap manipulating programs. In particular: (1) we identify the structural properties inferred by COSTA which need to be verified and extend the *Java Modeling Language* (JML) by suitable new constructs; (2) we extend the program logic used during verification by additional theories for structural heap properties including acyclicity or disjointness of heap regions. Extensive work with implementation and improvement of the proof-search strategies for the newly introduced theories was required to achieve a high degree of automation; (3) we formalize faithfully the notion of maximal path-length of an acyclic data structure in KeY's logic. This theory is equipped with lemmas that match the requirements of the path-length analysis performed in COSTA; and (4) realizing the cooperation between COSTA and KeY has required a number of non-trivial extensions of both systems.

The paper is organized as follows: Sec. 2 recalls the framework of [3]; Sec. 3 presents the additional components that need to be verified for carrying out the extension; Sec. 4 describes how the KeY logic has been extended to express and verify structural heap properties and path-length assertions; experimental results are presented in Sec. 5; and Sec. 6 concludes and discusses related work.

2 The Framework: Verification of Resource Guarantees

In this section we review the verification framework for upper bounds (UBs) as proposed in [3] which does not take heap-allocated data structures into account. Sec. 2.1 describes the components involved in a resource guarantees analysis while Sec. 2.2 details the formal verification of these components with KeY.

```
1 void scoreBoard(int[][] v) {
2   //@ ghost int v_len =v.length
3   int i=0, j=0;
4   //@ assert (j=0 ∧ i=0 ∧ v.length=v_len)
5   //@ ghost int i1=i, j1=j, v_len1=v.length
6   //@ ghost int i2=i, j2=j, v_len2=v.length
7   //@ decreases ( (v.length − i) ≥ 0 ? (v.length − i) : 0)
8   //@ loop_invariant (i2=0 ∧ i2=i ∧ v_len2≥0) ∨ (i2=0 ∧ i≥1 ∧ v_len2≥i)
9   while (i < v.length) {
10    j=0;
11    //@ assert (v_len1>i ∧ i1=i ∧ j=0)
12    //@ ghost int i3=i, j3=j, v_len3=v.length
13    //@ ghost int i4=i, j4=j, v_len4=v.length
14    //@ decreases ( (i − j) ≥ 0 ? (i − j) : 0)
15    //@ loop_invariant (j4=0 ∧ j4=j ∧ i4=i) ∨ (j4=0 ∧ j≥1 ∧ i4≥j ∧ i4=i)
16    while (j < i) {
17      v[i][j]=i + j;
18      j++;
19      //@ assert (j=j3+1 ∧ i3=i);
20      //@ set i3=i, j3=j, v_len3=v.length
21    }
22    i++;
23    //@ assert (v.length1>i ∧ i=i1+1)
24    //@ set i1=i, j1=j, v_len1=v.length
25  }
26 }
```

Fig. 1. COSTA's output for a simple example working on integer data

2.1 Inference of Resource Guarantees

Cost analyzers [1,2] usually infer UBs for each iterative and recursive construct (loops) and then compose the results in order to obtain UBs for the methods of interest. W.l.o.g., we focus on polynomial UBs which are the result of composing simple loops, but the same components are used to infer UBs for programs with logarithmic and exponential complexities. Intuitively, in order to infer an UB for a single loop, we infer an UB A on the worst-case cost of a single execution of its body and an UB I on the number of iterations that it can perform. Then, $A * I$ is an UB for the loop. To infer A and I COSTA relies on the program analysis components described below that provide the necessary information. The results are provided by COSTA as JML annotations that KeY will attempt to verify.

Ranking functions. For each loop, COSTA infers as UB on the number of iterations a linear function I from the loop variables to $\mathbb{N}$ which is strictly decreasing at each iteration. Ranking functions are of the form $\mathsf{nat}(\ell)$, where $\mathsf{nat}(\ell) = \max(0, \ell)$, which can be translated to the JML annotation "*//@ decreasing* $\ell > 0 \ ? \ \ell : 0$".

Example 1. Consider the method `scoreBoard()` given in Fig. 1, where two nested loops are used to initialize some matrix values. For the inner loop COSTA infers at line 14 the ranking function $f(i,j) = \texttt{nat}(i-j)$ which safely bounds the number of iterations. For the outer loop, the number of iterations is bounded by the ranking function that appears in line 7 which involves the length of the array.

Loop invariants. Loop invariants, together with size relations, are needed to compute the worst-case cost A of executing one loop iteration. For each loop in the program, COSTA infers an invariant φ that involves the loop variables $\bar{v}$ and auxiliary variables $\bar{w}$ such that each w_i represents the initial value of v_i. The JML annotation for this invariant consists of one line defining all $\bar{w}$ as ghost variables ("*//@ ghost int* $w_1 = v_1$*;...; int* $w_n = v_n$", lines 6, 13 in Fig. 1) and one line for the loop invariant ("*//@ loop_invariant* φ", lines 8, 15 in Fig. 1).

Example 2. Consider the invariant for the outer loop at line 8. The left disjunct corresponds to first visit to that program point, and the right disjunct to visit it after executing the loop body at least once. Note that separating the invariant into these two cases results in a more precise UB, and in addition helps KeY in verifying the invariant. We declared as ghost variable in line 6 such that i_2,j_2 and v_len_2 correspond to the initial value of `i`, `j` and `v.length` when entering the loop for the first time. The invariant states that `i` is always smaller than or equal to the initial value of `v.length` ($i \leq v\_len_2$) This is essential to bound the worst-case cost of the loop, since the cost of each iteration depends on `i`.

Size relations. Given a fragment of code (a scope), COSTA infers size relations between the values of the variables at a certain program point of interest within the scope and their initial values when entering the scope. This allows composing the cost of the different code fragments. In particular, for each loop (or method call), COSTA infers the relation φ between the values of variables before a loop (or call) entry and the entry of its parent scope. Suppose that the loop (or call) is at line L_l, its parent scope starts at line L_p, $\bar{v}$ are the variables of interest at line L_l, and $\bar{w}$ represent their values at line L_p. Then we add the JML annotation "*//@ ghost int* $w_1 = v_1$*;...; int* $w_n = v_n$;" immediately after line L_p to capture the values of $\bar{v}$ at line L_p, and the JML annotation "*//@ assert* φ" immediately before line L_l to state that the relation φ must hold at the program point.

Example 3. Let us demonstrate the need for size relations: (1) during cost analysis, the cost of the outer loop is inferred first in terms of the values of `i` and `v.length` before entering the loop, and later is transformed to be in terms of the length of the input array. For this, COSTA uses the size relation at line 4 which relates the values at that program point to those at line 2 using the corresponding ghost variables; (2) similarly, the cost of the inner loop is first inferred in terms of the values of `i` and `j` before entering the loop, and later is transformed to be in terms of their values when entering the outer loop. Assuming that i_1, j_1 and v_len_1 are respectively the value of `i`, `j`, and `v.length`, line 11 includes the size relation required to do such transformation. Note that since these code

fragments appear inside a loop, the values of i_1, j_1 and v_len_1 should be updated in each iteration. This is done by defining and initializing them at line 5 (for the first iteration) and modifying them in each iteration at the end of the loop (line 24). The size relation at line 23 is used by COSTA to synthesize a ranking function, this also helps KeY in proving that it is indeed a ranking function; and (3) lines 12, 19 and 20 encode the size relation of the inner loop.

Upper Bounds. In the verification phase it suffices to prove the correctness of the inferred ranking functions, loop invariants, and size relations: based on these, it is straightforward to compute an UB for the method by applying parametric integer programming (PIP) to obtain A and then just multiply $I * A$.

Example 4. We start from the innermost loop at line 16. Assuming that executing the condition costs (at most) c_1 instructions, and that the cost of each iteration (i.e., the loop body) is c_2 instructions, then it is clear that $\mathsf{nat}(i_4 - j_4) * (c_1 + c_2) + c_1$ is an UB on the cost of this loop. Next, we move to the outer loop at line 9. Let us assume that the cost of the comparison is (at most) c_3 instructions, the code at line 10 costs c_4 instructions, and the code at line 22 is c_5 instructions. Then, the cost of each iteration of this loop is $c_3 + c_4 + \mathsf{nat}(i_4 - j_4) * (c_1 + c_2) + c_1 + c_5$, where the highlighted subexpression is the cost of the inner loop. Note that each iteration might have a different cost, since $i_4 - j_4$ is not the same for all iterations. The solution is to find the worst-case cost A in terms of v_len_2, i_2, j_2 such that $A \geq i_4 - j_4$ in all iterations. Then, $\mathsf{nat}(v\_len_2 - i_2) * [c_3 + c_4 + \mathsf{nat}(A) * (c_1 + c_2) + c_1 + c_5] + c_3$ is an UB for the loop. To find such A, COSTA solves the PIP problem of maximizing the objective function $i_4 - j_4$ w.r.t. the loop invariant (line 8) and the size relations (line 11) where v_len_2, i_2, j_2 are the parameters. This produces an expression in terms of v_len_2, i_2, j_2 which is greater than or equal to $i_4 - j_4$ in all iterations of the loop. In our example, it is $A = v\_len_2 - 1$. We finally can compute the cost of the `scoreBoard` method. Assume that the cost of line 3 is c_6, then the cost of the method is $c_6 + \mathsf{nat}(v\_len_2 - i_2) * [c_3 + c_4 + \mathsf{nat}(v\_len_2 - 1) * (c_1 + c_2) + c_1 + c_5] + c_3$. We need to express this UB in terms of the input parameter v_len. For this, COSTA maximizes (using PIP) $v\_len_2 - i_2$ and $v\_len_2 - 1$ w.r.t. the size relation at line 4 and, respectively, obtains v_len and $v\_len - 1$. Therefore, $c_6 + \mathsf{nat}(v\_len) * [c_3 + c_4 + \mathsf{nat}(v\_len - 1) * (c_1 + c_2) + c_1 + c_5] + c_3$ is the UB for `scoreBoard`.

2.2 Verification by Symbolic Execution

The program logic used by KeY is *JavaCard Dynamic Logic* (JavaDL) [5], a first-order dynamic logic with arithmetic. JavaDL extends sorted first-order logic by a program modality $\langle\cdot\rangle\cdot$. Let p denote a sequence of executable Java statements and ϕ an arbitrary JavaDL formula, then $\langle\mathsf{p}\rangle\phi$ is a formula which states that program p terminates and in its final state ϕ holds. A typical formula looks like

$$\mathsf{i} \doteq i0 \wedge \mathsf{j} \doteq j0 \;\texttt{->}\; \overbrace{\langle \mathsf{i=j-i;j=j-i;i=i+j;} \rangle}^{\mathsf{p}} (\mathsf{i} \doteq j0 \wedge \mathsf{j} \doteq i0)$$

where i, j are program variables represented as *non-rigid* constants. Non-rigid constants and functions are state-dependent: their value can be changed by programs. The *rigid* constants $i0, j0$ are state-independent: their value cannot be changed. The formula above says that if program p is executed in a state where i and j have values $i0$, $j0$, then p terminates *and* in its final state the values of the variables are swapped. To reason about JavaDL formulas, KeY employs a sequent calculus whose rules perform *symbolic execution* of the programs in the modalities. Here is a typical rule:

$$\textsf{ifSplit}\ \frac{\Gamma, b \Longrightarrow \langle \{\textsf{p}\}\textsf{rest}\rangle\phi, \Delta \quad \Gamma, \neg b \Longrightarrow \langle \{\textsf{q}\}\textsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \textsf{if (b) \{p\} else \{q\} rest}\rangle\phi, \Delta}$$

As values are symbolic, it is in general necessary to split the proof whenever an implicit or explicit case distinction is executed. It is also necessary to represent the *symbolic* values of variables throughout execution. This becomes apparent when statements with side effects are executed, notably assignments. The assignment rule in JavaDL looks as follows:

$$\textsf{assign}\ \frac{\Gamma \Longrightarrow \{\textsf{x} := \textsf{val}\}\langle \textsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \textsf{x = val; rest}\rangle\phi, \Delta}$$

The expression in curly braces in the premise is called *update* and is used in KeY to represent symbolic state changes. An *elementary* update $loc := val$ is a pair of a program variable and a value. The meaning of updates is the same as that of an assignment, but updates can be composed in various ways to represent complex state changes. Updates u_1, u_2 can be composed into *parallel updates* $u_1 \| u_2$. In case of clashes (updates u_1, u_2 assign different values to the same location) a last-wins semantics resolves the conflict. This reflects left-to-right sequential execution. Apart from that, parallel updates are applied simultaneously, i.e., they do not depend on each other. Update application to a formula/term e is denoted by $\{u\}e$ and forms itself a formula/term.

Verifying Size Relations. JML annotations are proven to be valid by symbolic execution. For example, in the method scoreBoard() one starts with execution of the variable declarations. Ghost variable declarations and assignments to ghost variables (*//@ set var=val;*) are treated like Java assignments. If a JML assertion "assert φ;" is encountered during symbolic execution, the proof is split: the first branch must prove that the assertion formula φ holds in the current symbolic state; the second branch continues symbolic execution. In the scoreBoard example, a proof split occurs before entering each loop. This verifies the size relations among variables as derived by COSTA and encoded in terms of JML assertions.

Verifying Invariants and Ranking Functions. Verification of the loop invariants and ranking functions obtained from COSTA is achieved with a tailored loop invariant rule that has a variant term to ensure termination:

$$\textsf{loopInv}\ \frac{\begin{array}{ll}(i) & \Gamma \Longrightarrow Inv \wedge dec \geq 0, \Delta \\ (ii) & \Gamma, \{\mathcal{U}_A\}(b \wedge Inv \wedge dec \doteq d0) \Rightarrow \\ & \quad \{\mathcal{U}_A\}\langle \textsf{body}\rangle(Inv \wedge dec < d0 \wedge dec \geq 0), \Delta \\ (iii) & \Gamma, \{\mathcal{U}_A\}(\neg b \wedge Inv) \Longrightarrow \{\mathcal{U}_A\}\langle \textsf{rest}\rangle\phi, \Delta \end{array}}{\Gamma \Longrightarrow \langle \textsf{while (b) \{ body \} rest}\rangle\phi, \Delta}$$

Inv and dec are obtained, respectively, from the loop_invariant and decreasing JML annotations generated by COSTA. Premise (i) ensures that invariant Inv is valid just before entering the loop and that the variant dec is non-negative. Premise (ii) ensures that Inv is preserved by the loop body and that the variant term decreases strictly monotonic while remaining non-negative. Premise (iii) continues symbolic execution upon loop exit. The integer-typed variant term ensures loop termination as it has a lower bound (0) and is decreased by each loop iteration. Using COSTA's derived ranking function as variant term obviously verifies that the ranking function is correct. The update $\mathcal{U}_A$ assigns to all locations whose values are potentially changed by the loop a fixed, but unknown value. This allows using the values of locations that are unchanged in the loop during symbolic execution of the body.

Contracts. COSTA also infers *contracts* which specify pre- and post-conditions on the input and output arguments of each method. Contracts are useful for modular verification in KeY.

3 Upper Bounds for Heap Manipulating Programs

When input arguments of a method are of reference type, its UB is usually not specified in terms of the concrete values within the data structures, but rather in terms of some *structural* properties of the involved data structures. For example, if the input is a list, then the UB would typically depend on the length of the list instead of the concrete values in the list.

Example 5. Consider the program in Fig. 2 where class `List` implements a linked list as usual. For method `insert`, COSTA infers the UB $c_1 * \mathsf{nat}(x) + c_2$ where x refers to the length of `x`, and c_1/c_2 are constants representing the cost of the instructions inside/before & after the loop. The UB depends on the length of `x`, because the list is traversed at lines 16–19.

The example shows that cost analysis of heap manipulating programs requires inferring information on how the size of data structures changes during the execution, similar to the invariants and size-relations that are used to describe how the values of integer variables change. To do so, we first need to fix the meaning of "size of a data structure". We use the path-length measure which maps data structures to their *depth*, such that the depth of a cyclic data structure is defined to be ∞. Recall that the depth of a data structure is the maximum number of nodes (i.e. objects) on a path from the root to a leaf. Using this size measure, COSTA infers invariants and size relations that involve both integer and reference variables, where the reference variables refer to the depth of the corresponding data structures. Once the invariants are inferred, synthesizing the UBs follows the same pattern as in Sec. 2. In the following, we identify the essential information of the path-length analysis (and related analyses) that must be verified later by KeY.

```
1 //@ requires \acyclic(x)
2 //@ ensures \acyclic(\result)
3 //@ ensures \depth(\result) ≤ \depth(x) + 1
4 public static List insert(List x, int v) {
5     //@ ghost List x_0 = x;
6     List p = null;
7     List c = x;
8     List n = new List(v, null);
9     //@ ghost List c_0 = c
10    //@ assert \depth(n) = 1 ∧ \depth(c_0) = \depth(x_0)
11    //@ decreasing \depth(c)
12    //@ loop_invariant \depth(c_0) ≥ \depth(c)
13    //@ loop_invariant \acyclic(n) ∧ \acyclic(p) ∧ \acyclic(x) ∧ \acyclic(c)
14    //@ loop_invariant \disjoint({n, x})∧\disjoint({n, c})∧\disjoint({n, p})
15    //@ loop_invariant !\reachPlus(p, x) ∧ !\reachPlus(n, x) ∧ !\reach(n, p)
16    while ( c != null ∧ c.data < v) {
17        p = c;
18        c = c.next;
19    }
20    if ( p == null ) {
21        n.next = x;
22        x = n;
23    } else {
24        n.next = c;
25        p.next = n;
26    }
27    return x;
28 }
```

Fig. 2. The running example, with (partial) JML annotations

3.1 Path-Length Analysis

Path-length analysis is based on abstracting program states to linear constraints that describe the corresponding path-length relations between the different data structures. For example, the linear constraint $x < y$ represents all program states in which *the depth of the data structure to which* `x` *points is smaller than the depth of the data structure to which* `y` *points*. Starting from an initial abstract state that describes the path-length relations of the initial concrete state, the analysis computes path-length invariants for each program point of interest. In order to verify the path-length information with KeY, we have extended JML with the new keyword `\depth` that gives the depth of a data structure to which a reference variable points. In particular, for invariants, size-relations, and contracts, if the corresponding constraints include a variable x, corresponding to a reference variable `x`, we replace all occurrences of x by `\depth(x)`.

Example 6. We explain the various path-length relations inferred by COSTA for the method `insert` of Fig. 2, and how they are used to infer an UB. Due to space

limitations, we only show the annotations of interest. For the loop at lines 16–19, COSTA infers that the depth of the data structure to which `c` points decreases in each iteration. Since the depth is bounded by 0, it concludes that $\texttt{nat}(c)$ is a ranking function for that loop. As a part of the loop invariant, COSTA infers that $c_0 \geq c$ where c_0 refers to the depth of the data structure to which `c` points before entering the loop and c to the depth of the data structure to which `c` points after each iteration. Using this invariant, together with the knowledge that the depth of c_0 equals to the depth of `x`, we have that $c_1 * \texttt{nat}(x) + c_2$ is an UB for `insert` (since the maximum value of `c` is exactly `x`). Another essential relation inferred by the path-length analysis (captured in the `ensures` clause in line 3) is that the depth of the list returned by `insert` is smaller than or equal to the depth of `x` plus one. This is crucial when analyzing a method that uses `insert` since it allows tracking the size of the list after inserting an element.

Path-length relations are obtained by means of a fixpoint computation which (symbolically) executes the program over abstract states. As a typical example, executing `x=y.f` adds the constraint $x' < y$ to the abstract state if the variable `y` *points to an acyclic* data structure, and $x' \leq y$ otherwise. On the other hand, executing `x.f=y` adds the constraints $\bigwedge\{z' \leq z + y \mid$ `z` might share with `x`$\}$ if it is guaranteed that `x` does not become cyclic after executing this statement. This is because, in the worst case, `x` might be a leaf of the corresponding data-structure pointed to by `z`, and thus the length of its new paths can be longer than the old ones at most by y. Obviously, to perform path-length analysis, we require information on (a) whether a variables certainly points to an acyclic data structure; and (b) which variables might share common regions in the heap.

3.2 Cyclicity analysis

The cyclicity analysis of COSTA [9] infers information on which variables *may* point to (a)cyclic data structures. This is essential for the path-length analysis. The analysis abstracts program states to sets of elements of the form: (1) $\texttt{x}\rightsquigarrow\texttt{y}$ which indicates that starting from `x` one *may* reach (with at least one step) the object to which `y` points; (2) $\circlearrowleft^{\texttt{x}}$ which indicates that `x` *might* point to a cyclic data structure; and (3) $\texttt{x}\diamond\texttt{y}$ which indicates that `x` *might* alias with `y`.

Starting from an abstract state that describes the initial reachability, aliasing and cyclicity information, the analysis computes invariants (on reachability, aliasing and cyclicity) for each program point of interest by means of a fixpoint computation which (symbolically) executes the program instructions over the abstract states. For example, when executing `y=x.f`, then `y` inherits the cyclicity and reachability properties of `x`; and when executing `x.f=y`, then `x` becomes cyclic if before the instruction the abstract state included $\circlearrowleft^{\texttt{y}}$, $\texttt{y}\rightsquigarrow\texttt{x}$, or $\texttt{y}\diamond\texttt{x}$.

On the verification side, to make use of the inferred cyclicity relations, we extend JML by the new keyword `\acyclic` which *guarantees* acyclicity. In contrast to COSTA, JML and KeY use shape predicates with *must*-semantics. Acyclicity information is then added in JML annotations at entry points of contracts and loops where we specify all variables which are guaranteed to be acyclic. For loop

entry points as invariants (as in line 13) and for contracts as pre- and postconditions (as in lines 1, 2). To make use of the reachability relations we extend JML by the new keyword \reachPlus(x, y), which indicates that y *must* be reachable from x in at least one step, and use the standard keyword \reach(x, y) which indicates that y *must* be reachable from x in zero or more steps (i.e., they might alias). The *may*-information of COSTA about reachability and aliasing is then added as *must*-predicates in JML (in loop entries and contracts) as follows: let A be the set of judgments inferred by COSTA for a given program point, then we add !\reachPlus(x, y) whenever $\texttt{x} \rightsquigarrow \texttt{y} \notin A$, and we add !\reach(x, y) whenever $\texttt{x} \rightsquigarrow \texttt{y} \notin A \wedge \texttt{x} \diamond \texttt{y} \notin A$ (for example, in line 15).

3.3 Sharing analysis

Knowledge on possible sharing is required by both path-length and cyclicity analyses. The sharing analysis of COSTA is based on [15] where abstract states are sets of pairs of the form $\texttt{x} \bullet \texttt{y}$ which indicate that x and y might share a common region in the heap. The sharing invariants are propagated from an initial state by means of a fixpoint computation to the program points of interest. For example, when executing y=x.f, the variable y will only share with anything that shared with x (including x itself); on the other hand, when executing x.f=y, the variable x keeps its previous sharing relations, and in addition it might share with y and anything that shared with y before.

Obviously, KeY needs to know about the sharing information inferred by COSTA to verify acyclicity and path-length properties. To this end, we extended JML by the new keyword \disjoint which states that its argument, a set of variables, *does not share* any common region in the heap (for example, in line 14).

4 Verification of Path-Length Assertions

Structural heap properties, including acyclicity, reachability and disjointness, are essential both for path-length analysis and for the verification of path-length assertions. However, while the path-length analysis performed by COSTA maintains cyclicity and sharing, the complementary properties are used as primitives on the verification side. The reason is that the symbolic execution machinery of KeY starts with a completely unspecified heap structure that subsequently is refined using the inferred information about acyclicity and disjointness. In the following we explain how structural heap properties are formalized in the dynamic logic (JavaCard DL) used in this paper and implemented in KeY [5].

4.1 Heap Representation

First we briefly explain the logical modeling of the heap in JavaCard DL.[4] The heap of a Java program is represented as an element of type $Heap$. The $Heap$

[4] Note that this is *not* the heap model described in earlier publications on KeY such as [5]. In the present paper we use an explicit heap model based on [18].

data type is formalized using the theory of arrays and associates locations to values. A location is a pair (o, f) of an object o and a field f. The *select* function allows to access the value of a location in a heap h by $select(h, o, f)$. The complementary update operation which establishes an association between a location (o, f) and a value val is $store(h, o, f, val)$. To improve readability, when the heap h it is clear from the context, we use the familiar notation $o.f$ and $o.f := val$ instead of select and store expressions. Based on this heap model, we define a rule for symbolic execution of field assignments (cf. the assign rule in Sec. 2.2). It simply updates the global heap program variable with the updated heap object:

$$\textsf{assign}\ \frac{\Gamma \Longrightarrow \{\mathsf{heap} := store(\mathsf{heap}, \mathsf{o}, \mathsf{f}, \mathsf{v})\}\langle \mathsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{o.f} = \mathsf{v};\ \mathsf{rest}\rangle\phi, \Delta}$$

4.2 Predicates for Structural Heap Properties

For the sake of readability, in Sec. 3, we gave simplified versions of the predicates \depth, \acyclic, \reach, \reachPlus and \disjoint as compared to the actual implementation. In reality, these predicates have an extra argument that restricts their domain to a given set of fields. For example, instead of $\texttt{\textbackslash depth}(x)$ we might actually have $\texttt{\textbackslash depth}(\{x.\texttt{next}\}, x)$ which refers to the depth of x considering only those paths that go through the field next. A syntactic analysis infers automatically a safe approximation of these sets of fields by taking the fields explicitly used in the corresponding code fragment.

Ultimately, the various structural heap properties are reduced to reachability between objects which, therefore, must be expressible in the underlying program logic. The counterpart of JML's \reach predicate in JavaCard DL is

$$\texttt{\textbackslash reach} : Heap \times LocSet \times Object \times Object \times int$$

and expresses *bounded reachability* (or n-reachability): an object e is n-reachable from an object s with respect to a heap h and a set of locations l (of type $LocSet$) if and only if there exists a sequence $s = o_1 o_2 \cdots o_n = e$ where $o_{i+1} = o_i.f_i$ and $(o_i, f_i) \in l$ for all $0 < i < n$. The predicate $\texttt{\textbackslash reach}(h, l, s, e, n)$ is formally defined as $n \geq 0 \wedge s \neq \mathsf{null} \wedge ((n \doteq 0 \wedge s \doteq e) \vee \exists f.(o, f) \in l \wedge \texttt{\textbackslash reach}(h, l, s.f, e, n-1))$. As a consequence, from null nothing is reachable and also null cannot be reached.

Location sets in JavaCard DL are formalized in the data type $LocSet$ which provides constructors and the usual set operations (see [18] for a full account). Here we need only three location set constructors: the constructor *empty* for the empty set, the constructor $singleton(o, f)$ which takes an object o and a field f and constructs a location set with location (o, f) as its only member, and the constructor $allObjects(f)$ which stands for the location set $\{(o, f) \mid o \in Object\}$.

Example 7. $\texttt{\textbackslash reach}(h, allObjects(\mathsf{next}), head, last, 5)$ is evaluated to true iff the object *last* is reachable from object *head* in five steps by a chain of next fields.

Based on \reach we could directly axiomatize structural heap predicates such as $\texttt{\textbackslash acyclic}(h, l, o)$ or $\texttt{\textbackslash disjoint}(h, l, o, u)$. Instead we prefer to reduce structural heap predicates to $\texttt{\textbackslash reachPlus}(h, l, o, u)$ which is the counterpart of

the JML function of the same name in Sec. 3.2 and expresses reachability in at least one step. This has several advantages over using \reach: (1) the definition of predicates such as \acyclic does not use the step parameter of the \reach predicate and one would use existential quantification to eliminate it which impedes automation; and (2) for $\texttt{\reachPlus}(h, l, o, u)$ to hold one has to perform at least one step using a location in l. This renders the definition of properties such as \acyclic less cumbersome as the zero step case has been excluded.

The predicate \reachPlus can defined with the help of \reach and this definition can be used if necessary, however, in the first place we use a separate axiomatization of \reachPlus. This helps to avoid (or at least to delay as long as possible) the reintroduction of the step parameter and, hence, an additional level of quantification. For space reasons, we do not give the calculus rules for the axioms and auxiliary lemmas of the structural heap predicates like \acyclic and \disjoint (which are not too surprising). Instead, we describe in the following section one central difficulty that arises when reasoning about structural heap properties and how we solved it to achieve higher automation.

4.3 Field Update Independence

When reasoning about structural heap predicates one often ends up in a situation where one has to prove that a heap property is still valid after updating a location on the heap, i.e, after executing one or several field assignments. For instance, we might know that $\texttt{\acyclic}(h, l, u)$ holds and have to prove that after executing the assignment o.f=v; the formula $\texttt{\acyclic}(\mathit{store}(h, o, f, v), l, u)$ holds.

A precise analysis of the effect of a field update is expensive and makes automation significantly harder. As it is common in this kind of situation, it helps to *optimize the common case*. In the present context, this means to decide in most cases efficiently that a field assignment does not effect a heap property at all. This is sufficiently achieved by two simple checks:

1. The expression $\mathit{singleton}(o, f) \subseteq l$ checks whether an updated location $o.f$ is in the location set l of the heap property to be preserved. This turns out to be inexpensive for most (if not all) practically occurring cases. Whenever this check fails, the resulting *store* can be removed from the argument of the heap property. For instance, an assignment o.data=5 to the data field of a list does not change the list structure which depends solely on the next field. In that case we can rewrite $\texttt{\acyclic}(\mathit{store}(h, \mathsf{o}, \mathsf{data}, 5), l, u)$ to $\texttt{\acyclic}(h, l, u)$.
2. To check whether an object o whose field has been updated is reachable from one of the other mentioned objects, is more expensive than the previous one, but still cheaper than a full analysis. For example, we can check whether the object o is reachable from object u in case of $\texttt{\acyclic}(\mathit{store}(h, o, f, v), l, u)$. If the answer is negative we can again discard the store expression.

4.4 Path-Length Axiomatization

In general, the JML assertions generated by COSTA refer to the path-length of a data structure o as $\texttt{\depth}(l, o)$ where l is the location set restricting the depth

Bench	Certificate Generation				Cert. Size		Generation/Checking		
	$\mathbf{T}_{heap}$	$\mathbf{T}_{ana}$	$\mathbf{T}_{jml}$	$\mathbf{T}_{ver}$	**Nod**	**Br**	$\mathbf{T}_{gen}$	$\mathbf{T}_{check}$	%
`traverse`	14	36	2	2300	1208	52	2338	1100	47.05
`create`	54	150	8	3100	1499	47	3258	1400	42.97
`insert`	282	374	16	40800	19252	636	41190	5800	14.08
`indexOf`	26	86	4	5900	2439	67	5990	1800	30.05
`reverse`	72	130	8	20900	14206	673	21038	3400	16.16
`array2List`	62	154	8	2600	1457	37	2762	1400	50.69
`copy`	76	132	10	22600	14147	673	22742	3100	13.63
`searchtree`	142	202	6	3700	2389	97	3908	1500	38.38

Table 1. Statistics for the Generation and Checking of Resource Guarantees

to certain locations. This JML function is mapped to the JavaCard DL function $\texttt{\textbackslash depth}(h, l, o)$ which is evaluated to the maximal path-length of o in heap h using only locations from l. Its axiomatization is based on the n-reachability predicate `\reach` expressing that there exists an object u reachable in $\texttt{\textbackslash depth}(h, l, o)$ steps and that there is no object z reachable from o in more than $\texttt{\textbackslash depth}(h, l, o)$ steps. This definition is not used by default by the theorem prover, instead, automated proof search relies mainly on a number of lemmas that state more useful higher-level properties. For instance, given a term like $\texttt{\textbackslash depth}(\mathit{store}(h, o, f, v), l, u)$ there is a lemma which checks that o is reachable from u and some acyclicity requirements. If that is positive then the lemma allows us to use the same approximation for `\depth` in case of a heap update as detailed in Sec. 3.1.

5 Experimental Results

The implementation of our approach required the following non-trivial extensions to COSTA and KeY: (1) generate and output in COSTA the JML annotations `\depth`, `\acyclic` and `\disjoint` so that KeY can parse them; (2) synthesize suitable proof obligations in JavaCard DL that ensure correctness of the resource analysis; (3) axiomatize the JML `\depth`, `\acyclic` and `\disjoint` functions in KeY as described in Sec. 4 and implement heuristics for automation; and (4) implement heuristic checks in KeY that allow fast verification of the common case as described in Sec. 4.4. The resulting extended versions of KeY and COSTA are available for download from `http://fase2012.hats-project.eu`.

Table 1 shows first experiments using a set of representative programs that perform common list operations as well as searching for an element in a binary tree. The experiments were performed using an Intel Core2 Duo at 2.53GHz with 4Gb of RAM running Linux 2.6.32. Columns $\mathbf{T}_{heap}$, $\mathbf{T}_{ana}$ and $\mathbf{T}_{jml}$ show, respectively, the times (in milliseconds) taken by COSTA to perform the heap analysis (cyclicity, sharing and path-length), to execute the whole analysis (heap and other analyses performed by COSTA), and to generate the JML annotations. Column $\mathbf{T}_{ver}$ shows the time taken by KeY to verify the JML annotations generated by COSTA. The size of the generated proofs is indicated by their number of nodes

Nod and branches **Br**. Column $\mathbf{T}_{gen}$ shows the total time taken to generate the proof ($\mathbf{T}_{ana}$+$\mathbf{T}_{jml}$+$\mathbf{T}_{ver}$) and $\mathbf{T}_{check}$ shows the time taken by KeY to check the validity of the proof. The last column (%) shows the ratio $\mathbf{T}_{check}/\mathbf{T}_{gen}$.

Our preliminary experiments show already that a proof-carrying code approach to resource guarantees can be realized using COSTA and KeY with both certificate generation and checking being fully automatic. In our framework the code originating from an untrusted *producer* should be bundled with the proof generated by COSTA + KeY for a given resource consumption. Then the code *consumer* can check locally and automatically with KeY whether the claimed resource guarantees are verified. As expected, checking an existing proof with KeY takes on average only around 30% of the time to produce it.

6 Conclusions and Related Work

This paper describes the combination of a state-of-the-art resource analyzer (COSTA) and a formal verification tool (KeY) to automatically infer *and verify* resource guarantees that depend on the size of heap-allocated data structures in Java programs. The distribution of work among the two systems is as follows: COSTA generates ranking functions, invariants, as well as size relations, and outputs them as extended JML annotations of the analyzed program; KeY then verifies the resulting proof obligations in its program logic and produces proof certificates that can be saved and reloaded.

Many software verification tools including KeY [5], Why [8], VeriFast [16], or Dafny [12] rely on automatic theorem proving technology. While most of these systems are expressive enough to model and prove heap properties of programs, such proofs are far from being automatic. The main reason is that functional verification of heap properties requires complex invariants that cannot be found automatically. In addition, automated reasoning over heap-allocated symbolic data is far less developed than reasoning over integers or arrays.

With this paper we show that the automation built into a state-of-the-art verification system is sufficient to reason successfully about resource-related heap properties. The main reasons for this are: (a) the required invariants are inferred automatically in the resource analysis stage; (b) a limited and carefully axiomatized signature for heap properties expressed in logic is used. This confirms the findings of the SLAM project [4] that existing verification technology can be highly automatic for realistic programs and a restricted class of properties.

There exist several other cost analyzers which automatically infer resource guarantees for different programming languages [10,11]. However, none of them formally proves the correctness of the upper bounds they infer. An exception is [6], which verifies and certifies resource consumption (for a small programming language and not for heap properties). For the particular case of memory resources, [7] formally certifies the correctness of the static analyzer. We have taken the alternative approach of certifying the correctness of the upper bounds that the tool generates. This is not only much simpler, but has the additional advantage that the generated proofs can act as resource certificates.

Acknowledgments: This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, the Spanish Ministry of Science and Innovation (MICINN) under th TIN-2008-05624 *DOVES* project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. *Proc. of ESOP'07*, vol. 4421 of *LNCS*, pp. 157–172. Springer, 2007.
3. E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. Verified Resource Guarantees using COSTA and KeY. In *Proc. of PEPM 2011*, pp. 73–76. ACM Press, 2011.
4. T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The Static Driver Verifier research platform. In *Proc. of CAV'10*, vol. 6174 of *LNCS*, pp. 119–122. Springer, 2010.
5. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, vol. 4334 of *LNCS*. Springer, 2006.
6. K. Crary and S. Weirich. Resource Bound Certification. In *POPL'05*, pp. 184–198. ACM Press, 2000.
7. J. De Dios and R. Peña. Certification of Safe Polynomial Memory Bounds. In *Proc. of FM'11*, LNCS. Springer, June 2011. To Appear.
8. J. C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pp. 173–177. Springer, 2007.
9. S. Genaim and D. Zanardini. The Acyclicity Inference of COSTA. In *Workshop on Termination (WST'10)*, July 2010.
10. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *Proc. of POPL'09*, pp. 127–139. ACM, 2009.
11. J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *Proc. of ESOP'10*, vol. 6012 of *LNCS*, pp. 287–306. Springer, 2010.
12. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. of LPAR'10*, vol. 6355 of *LNCS*, pp. 348–370. Springer, 2010.
13. G. Necula. Proof-Carrying Code. In *POPL 1997*. ACM Press, 1997.
14. A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Proc. of TACAS'98*, vol. 1384 of *LNCS*, pp. 151–166. Springer, 1998.
15. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Proc. of SAS'05*, vol. 3672 of *LNCS*, pp. 320–335. Springer, 2005.
16. J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *Proc. of FASE'08*, vol. 4961 of *LNCS*, pp. 261–275. Springer, 2008.
17. F. Spoto, F. Mesnard, and É. Payet. A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3), 2010.
18. B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, KIT, 2011.