Software and Systems Modeling manuscript No.
(will be inserted by the editor)

A Formal Verification Framework for Static Analysis*

As well as its instantiation to the resource analyzer COSTA and formal verification tool KeY

Elvira Albert · Richard Bubel · Samir Genaim · Reiner Hähnle · Germán Puebla · Guillermo Román-Díez

Received: date / Accepted: date

Abstract Static analysis tools, such as resource analyzers, give useful information on software systems, especially in real-time and safety-critical applications. Therefore, the question of the reliability of the obtained results is highly important. static analyzers typically combine a range of complex techniques, make use of external tools, and evolve quickly. To formally verify such systems is not a realistic option. In this work we a different approach whereby, instead of the *tools*, we formally verify the *results* of the tools. The central idea of such a formal verification framework for static analysis is the method-wise translation of the information about a program gathered during its static analysis into specification contracts that contain enough information for them to be verified automatically. We instantiate this framework with COSTA, a state-of-the-art static analysis system for sequential Java programs, for producing resource guarantees and KeY, a state-of-the-art verification tool, for formally verifying the correctness of such resource guarantees. *Resource guarantees* allow to be certain that programs will run within the indicated amount of resources, which may refer to memory consumption, number of instructions executed, etc. Our results show that the proposed tool cooperation can be used for automatically producing verified resource guarantees.

Keywords Cost Analysis · Closed-form Upper Bounds · Resource Analysis · Resource Guarantees

* This work is a revised and extended version of two conference papers published in the proceedings of PEPM'11 [8] and FASE'12 [9].

E. Albert (elvira@sip.ucm.es), S. Genaim (samir@fdi.ucm.es)
DSIC, Complutense University of Madrid (UCM), 28040 Madrid, Spain

R. Bubel (bubel@cs.tu-darmstadt.de), R. Hähnle (haehnle@cs.tu-darmstadt.de)
CSD, Technical University of Darmstadt, 64289 Darmstadt, Germany

G. Puebla (german@fi.upm.es), G. Román-Díez (groman@fi.upm.es)
DLSIIS, Technical University of Madrid (UPM), 28660 Boadilla del Monte (Madrid), Spain

1 Introduction

1.1 Overview

Static analysis tools, such as resource analyzers [3,23], termination checkers [1, 18], and verifiers [13,16] are by now powerful enough to be used on commercial software. Many of their potential application scenarios include safety- and security-critical aspects, therefore, the question of the reliability of the obtained results is highly important.

static analyzers typically combine a range of complex techniques such as symbolic computation, symbolic execution, abstract interpretation, SMT solving, various arithmetic solvers, etc. They make use of several external tools and evolve as quickly as their target scenarios. To formally verify such systems is not a realistic option. A more viable alternative is to construct a validating tool [30] which, after every run of the static analyzer, formally confirms that the results are correct and, optionally, generates correctness proofs. Such proofs could then be translated into *certificates*, following the proof-carrying code paradigm [20,29].

In this work, we focus on one specific class of analyzers that produce *resource guarantees*. The current dual trend of having elastic computational resources in the cloud and concurrent execution on many-core architectures make resource analysis a particularly interesting subdomain of static analysis. We show that the resulting information from a resource analyzer is sufficient to permit *fully automatic* formal verification of the obtained resource bounds with a software verification system. We demonstrate our approach with a pair of concrete systems: the COSt and Termination Analyzer for sequential Java bytecode (and hence Java) COSTA and the Java source code verification tool KeY. While this shows the viability of our approach, it is far from being specific for the pair COSTA/KeY. The general principle can be easily carried over to other combinations of static and formal verification :

1. Translate the result of the static analyzer at the granularity of a method (procedure) in terms of a *contract*, in our case a *resource contract*. Contracts are written in the specification language supported by the formal verification system, in our case JML (Java Modeling Language). The translation is generic and fully automatic. The contract-based approach yields verification tasks of manageable size.
2. Extract from the static analyzer not only the final result, but also invariants and auxiliary information that during the analysis. These become part of the contracts and are crucial for automation, because a formal verification system could not infer them.
3. Use the verification system to formally verify the target program against the contracts. This formally proves the validity of the results of the static analysis which are contained in the contracts as part of the postcondition.

This general approach is possible for any static analyzer that permits extraction of intermediate results and any formal verification system that has a contract language of sufficient expressivity. In addition to the steps outlined above, some engineering is necessary to achieve full automation of formal verification, but the overall approach is completely generic.

1.2 COSTA and KeY

COSTA receives as input the bytecode of a Java program, the signature of the method whose cost is to be inferred, a choice of one among several available cost models (termination [1], number of bytecode instructions [5], memory consumption [10], or number of calls to some method of interest) and automatically infers an *upper bound* (UB for short) on the cost as a function of the method's input arguments. The most challenging step is to infer UBs for the loops in the program [3]. Intuitively, this requires (1) bounding the number of iterations of each loop and (2) finding the worst-case cost among all iterations. *Ranking functions* [31] give us safe approximations for requirement (1). To infer the maximal cost in requirement (2), we need to track how the values of variables change at the loop iterations and the inter-relations between (the values of) variables. As we will see, this information is obtained in COSTA by means of *loop invariants* and *size relations*. The analysis algorithms used in COSTA for inferring the three main components of the UB generation (i.e., ranking functions, loop invariants, and size relations) were proven correct at a theoretical level [3]. However, there is no guarantee that correctness is preserved in the actual implementation, which is rather involved and includes translation an intermediate language.

KeY [15] is a source code verification tool for the Java programming language. Its coverage of Java is comparable to that of COSTA (nearly full sequential Java, plus a simplified concurrency model). KeY implements a logic-based setting of symbolic execution that allows deep integration with aggressive first-order simplification. While the degree of automation of KeY is very high on loop- and recursion-free programs, the user must in general supply suitable invariants to deal with loops and recursion. In general, invariants that are sufficient to prove complex functional properties cannot be inferred automatically. However, simpler invariants that are sufficient to establish UBs *can* be automatically derived in many cases and this is exactly COSTA's forte. Our work is based on the insight that the static analysis tool COSTA and the formal verification tool KeY have complementary strengths: COSTA is able to derive UBs of Java programs including the invariants needed to obtain them. This information is enough for KeY to *prove* the validity of the bounds and provide a certificate.

It is important to note that in the intermediate representation of COSTA, loops and recursion are represented in a uniform way and thus recursive programs are handled as well. In this case ranking functions are used to bound the depth of the recursion. For the sake of simplifying the presentation, in this article we focus only on loops. In Section 7 we comment briefly on how one can handle recursive programs.

1.3 Summary of Contributions

The overall contribution of this article is a practical framework to generate *verified resource guarantees*. The framework combines the complementary capabilities of static analyzers and formal verification tools. It is instantiated with two state-of-the-art tools, (1) COSTA, which is able to derive UBs for Java bytecode programs and (2) KeY, whose forte is to formally prove the validity of the bounds and provide a certificate of this proof. Our approach is developed for sequential Java programs.

We introduce our techniques in several steps, moving from simple to more complex resource bounds. Specifically, we start with bounds that depend only on Integer data, we then proceed to bounds that depend on the size of heap-allocated data and then we consider bounds that depend on shared mutable data. Technically, the presentation of our approach goes as follows:

1. We first consider programs whose resource consumption depends on Integer local variables. In this context, we identify those components of the UB that must be verified to guarantee the correctness of the UB and describe the verification techniques used to prove their correctness.
2. Next we consider programs whose resource consumption is bounded by the size of heap allocated data structures. We first identify the structural properties inferred by COSTA which need to be verified and extend the Java Modeling Language (JML) by suitable new constructs. Then 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. Finally we consider programs whose UB depends on (the size of) shared mutable data stored in object fields or array elements. The inference of UBs is based on the notion of *locality* which guarantees the heap accesses can be soundly transformed into *ghost* local variables. Then, UBs and JML annotations are derived by using the transformed program. The annotated program is then passed to KeY which generates a proof obligation expressing that the program satisfies the provided specification. That formula is loaded into the KeY theorem prover and discharged automatically.
4. Realizing the cooperation between COSTA and KeY required a number of non-trivial extensions of both systems (for example, COSTA works at the level of bytecode while KeY works on Java source code).

This article extends and improves two conference papers presented at [8] and [9]. The main technical extension is the enlargement of the class of programs that can be handled to include those whose resource consumption depends on shared mutable data.

1.4 Organization of the Article

The article is organized as follows. In Section 2 the core verification framework is presented, which is able to generate resource guarantees for Integer manipulating programs. This verification process is based on the formal verification of the components inferred by COSTA for generating UBs, including the ranking functions, loop invariants and size relations. For this purpose, COSTA generates JML annotations at appropriate locations in the Java source code. The resulting annotated Java source code is read by KeY to verify the correctness of the JML annotations with respect to the Java source code.

In Section 3 the core framework is extended to cover programs whose resource consumption is bounded by the size of heap-allocated data structures. Bounding their size requires to perform a number of structural heap analyses whose results,

inferred by COSTA, must be verified by KeY to guarantee the correctness of the UB.

In Section 4 the framework is further extended to obtain resource guarantees for programs whose resource consumption depends on *shared mutable data* stored in object fields or array elements. That section describes how COSTA performs a heap-sensitive resource analysis and how KeY verifies the results of such an analysis.

We report on a prototype implementation of our verification framework in Section 5 where also preliminary experimental results are presented. Finally, an overview on related work is given in Section 6, conclusions and a number of directions for future work are found in Section 7.

2 Verified Upper Bounds for Programs with Integer Data

We now present the different pieces of information that the resource analysis infers to obtain UBs and we describe the verification techniques used to prove their correctness. We first consider programs whose resource consumption only depends on integer data that is local to the corresponding method (i.e., not stored in the heap). As we have commented in the introduction, in the rest of this article we assume that the program under consideration is recursion-free, and later, in Section 7, we describe how such recursive programs can be handled.

2.1 Inference of Upper Bounds

We start by describing the techniques used in resource analysis for automatically inferring UBs. Cost analyzers [3,6] usually infer UBs for each iterative scope (loops) and then compose the results 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.

Example 1 On the left of Figure 1, we show a Java program, `insertion_sort(int[] A)`, that implements an insertion sort algorithm. The program contains two nested loops, the outer loop (also denoted by the *scope* s_{ol}), line 4 (abbreviated as L4), and the inner loop (scope s_{il}), which starts at L7. The program includes a method `decrement` that returns the value of the input argument minus one. In addition, we have scope s_0, which represents the instructions before calling the outer loop. We will show all the pieces of the UB inference on this example. ■

Note that in the above example we used the term *scope* to identify the components of a structured code, namely: (1) code fragments that correspond to loops; and (2) code fragments that appear before or after a loop (i.e., code that does not involve loop constructs). Note also that nested loops induce nested scopes. This term will be used in the rest of this article.

The computation of the UBs starts by compiling the bytecode program into a rule-based intermediate representation (RBR) of the program which preserves the cost information (all details can be found in [6]). The RBR program is obtained by building the *control flow graph* (CFG) of the bytecode program and representing each block in the CFG by means of a rule in the RBR. In a RBR program,

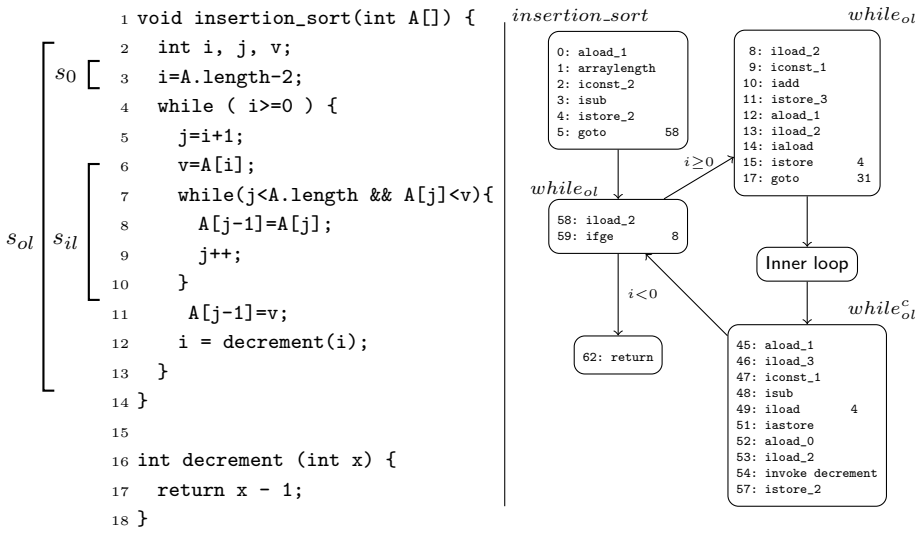

Fig. 1 Java source code (left) and CFG (right) for a simple example working on integer data

recursion[1]becomes the only form of iteration and conditional constructs are implemented by means of guarded rules. There are only *local variables* and there is no *stack*. Stack variables in bytecode are represented as local variables in the RBR. Each RBR rule has the form $q(\overline{x}, \overline{y})$, where q is the name of the rule, $\overline{x}$ are the input parameters and $\overline{y}$ are the output parameters of the rule. As we can have expressions of the form `i = i + 1`, where we want to capture that *the value of i after executing the instruction is equal to the value of before the instruction plus 1*, we apply a *single static assignment* (SSA) transformation that produces $i' := i + 1$, where i' represents the value of i after executing the instruction. Each instruction in the bytecode program is represented by one equivalent instruction in the RBR program. In the instructions, we store the program counter of each bytecode instruction and keep the same variable names as they have in the bytecode program. This information will be used for generating and printing the JML annotations of the source code.

Example 2 On the right of Figure 1 we show the CFG of the bytecode program on the left. The blocks that correspond to the inner loop are not shown as they are analogous. From the CFG of the program we generate its RBR by representing each block in the CFG as an RBR rule. For example, on the left of Figure 2 we show the RBR rules for the outer loop. It can be seen that this RBR captures all information in the bytecode, such as loop conditions, stack variables and variable identifiers. The instruction $s_1' := s_1 + s_2$ that corresponds to the bytecode `10: iadd`, exemplifies the SSA transformation, because s_1' represents the value of s_1 after executing the instruction. The control flow instructions, such as `goto` or `ifge`, are wrapped with a `nop` such that their cost can be considered later.

[1] Recall that we assume that the Java program is recursion-free, thus "recursion" in the RBR actually correspond only to loop constructs in the Java program.

$while_{ol}(\langle l_0,l_1,l_2,l_3,l_4\rangle,\langle l_2,l_3,l_4\rangle) \leftarrow i<0,$
$\quad s_1 := l_2,$ (58: iload_2)
$\quad$ nop(ifge) (59: ifge 8)

$while_{ol}(\langle l_0,l_1,l_2,l_3,l_4\rangle,\langle l_2,l'_3,l'_4\rangle) \leftarrow i\geq 0,$
$\quad s_1 := l_2,$ (8: iload_2)
$\quad s_2 := 1,$ (9: iconst_1)
$\quad s'_1 := s_1 + s_2,$ (10: iadd)
$\quad l'_3 := s'_1,$ (11: istore_3)
$\quad s''_1 := l_1,$ (12: aload_1)
$\quad s'_2 := l_2,$ (13: iload_2)
$\quad s''_1 := s''_1[s'_2],$ (14:iaload)
$\quad l'_4 = s'''_1$ (15: istore 4)
$\quad$ nop(goto), (17: goto 31)
$\quad while_{il}(\langle l_1,l_3,l'_4\rangle,\langle l'_3\rangle).$

$while^c_{ol}(\langle l_0,l_1,l_2,l_3,l_4\rangle,\langle l''_2,l'_3,l'_4\rangle) \leftarrow$
$\quad s_1 := l_1,$ (45: aload_1)
$\quad s_2 := l_3,$ (46: iload_3)
$\quad s_3 := 1,$ (47: iconst_1)
$\quad s'_2 := s_2 - s_3,$ (48: isub)
$\quad s'_3 := l_4,$ (49: iload 4)
$\quad s_1[s'_2] := s'_3,$ (51: iastore)
$\quad s'_1 := l_0,$ (52: aload_0)
$\quad s''_2 := l_2,$ (53: iload_2)
$\quad decrement(\langle s'_1,s''_2\rangle,s''_1)$ (54: invoke)
$\quad l'_2 := s''_1,$ (57:istore_2)
$\quad while_{ol}(\langle l_0,l_1,l'_2,l_3,l_4\rangle,\langle l''_2,l'_3,l'_4\rangle)$

s_0
$\quad insertion\_sort(\langle A_0\rangle,\langle\rangle) \leftarrow$
$\quad\quad i := A-2,$
$\quad\quad while_{ol}(\langle A,i,0,0\rangle,\langle i,j,v\rangle)).$

s_{ol}
$\quad while_{ol}(\langle A,i,j,v\rangle,\langle i,j,v\rangle) \leftarrow i<0.$
$\quad while_{ol}(\langle A,i,j,v\rangle,\langle i',j''',v''\rangle) \leftarrow i\geq 0,$
$\quad\quad j' := i+1,$
$\quad\quad v' := A[i],$
$\quad\quad while_{il}(\langle A,j',v'\rangle,\langle j''\rangle),$
$\quad\quad while^{co}_{ol}(\langle A,i,j'',v'\rangle,\langle i',j''',v''\rangle).$
$\quad while^{co}_{ol}(\langle A,i,j,v\rangle,\langle i'',j',v'\rangle) \leftarrow$
$\quad\quad A[j-1] := v,$
$\quad\quad decrement(\langle i\rangle,\langle i'\rangle),$
$\quad\quad while_{ol}(\langle A,i',j,v\rangle,\langle i'',j',v'\rangle).$

s_{il}
$\quad while_{il}(\langle A,j,v\rangle,\langle j\rangle) \leftarrow j\geq A.$
$\quad while_{il}(\langle A,j,v\rangle,\langle j''\rangle) \leftarrow j<A,$
$\quad\quad A[j-1] := A[j],$
$\quad\quad j' := j+1,$
$\quad\quad while_{il}(\langle A,j',v\rangle,\langle j''\rangle)$

$\quad decrement(\langle x\rangle,\langle r\rangle) \leftarrow$
$\quad\quad r = x-1.$

Fig. 2 Detailed RBR for the outer loop (left) and simplified RBR (right) for the running example

The RBR shown on the left of Figure 2 is the one used by the analyzer; however, for readability, we use a simplified version of it in our example (on the right of Figure 2) that contains the source code expressions instead of their corresponding bytecode instructions. In this simplified version we do not use the stack variables nor the `this` reference (l_0 in the RBR to the left). In addition, as the size analysis of COSTA uses the array length as size abstraction for arrays [6,34], we cannot observe the values of the arrays elements (as they have been abstracted away). Thus, the simplified RBR does not include loop guard expressions that access array elements, e.g., we do not include the condition `a[j]<v` in the simplified RBR nor in the remaining examples. It can be seen that loops, represented by rules $while_{il}$ and $while_{ol}$, are in recursive form in the RBR, and the termination conditions of loops become the rule guards. The inner loop is defined by the rules with head $while_{il}$ and the loop condition is evaluated by means of the guarded rules $j \geq A$ and $j < A$. Similarly, the outer loop is compiled into the rules $while_{ol}$ and $while^c_{ol}$. It is easy to see that the scopes s_0, s_{ol}, s_{il} in the source code of Figure 1, can also be identified at the level of the RBR – each scope is typically defined by a set of rules. ■

The RBR rules are transformed into direct recursive form [6] by a simple unfolding transformation. In the example, the rule $while^c_{ol}$ is unfolded into the body of $while_{ol}$ and, thus, $while_{ol}$ becomes directly recursive. From such an RBR, the

cost analyzer generates its *cost relation system* (CRS), a set of cost equations that define the cost of executing the program w.r.t. any input data (see [6]). Consider an RBR rule of the form $C(\overline{x},\overline{z}) \leftarrow g, b_1, \ldots, b_n, Y_1(\overline{y}_1,\overline{z}_1), \ldots, Y_n(\overline{y}_n,\overline{z}_n), C(\overline{x}',\overline{z}')$, where $\overline{x}$ and $\overline{z}$ are respectively the input and the output arguments of the rule, g is the guard and $b_1, \ldots, b_n$ are bytecode instructions, $Y_1(\overline{y}_1), \ldots, Y_n(\overline{y}_n)$ are invocations of other rules, and $C(\overline{x}',\overline{z}')$ is a recursive call. The associated cost equation is of the form:

$$\langle C(\overline{x}) = expr + \sum_{i=1}^{k} Y_i(\overline{y}_i) + C(\overline{x}'), \varphi \rangle$$

where $\overline{x}$ are the input arguments of equation C and *expr* captures the cost of the instructions $g, b_1, \ldots, b_n$ w.r.t. a *cost model* that will be defined below. Further, $Y_i(\overline{y}_i)$ represent the cost of the calls to other equations using $\overline{y_i}$ as input arguments and $C(\overline{x}')$ the cost of the recursive call, and φ corresponds to the *size relations*, which are a set of linear constraints on variables $\overline{x}$ and $\overline{y}_i \cup \overline{y}$. Observe that the output variables are not arguments of the equations because the cost is a function of the input data only. However, they play an important role when computing the size relations, as we will see later. Rules that are not recursive or do not have calls to other rules are handled in the same way, but lack some elements in the equations. Let us explain the two crucial elements for the computation of the CRS, the *cost model* and the *size relations*.

Cost Models. A cost model is a function of the form $\mathcal{M}(b) \rightarrow \mathbb{Z}$, which returns an integer number corresponding to the cost of executing the bytecode instruction b. As an example, if we are interested in counting the number of executed instructions, then $\mathcal{M}_i(b) = \mathcal{M}(b) = 1$ for each instruction. If we want to count the amount of memory allocated, then $\mathcal{M}_m(b) = \mathcal{M}(b) = size(b)$ if b is an instruction of the form $x = new\ C(\overline{y})$ and $size(b)$ is the amount of memory allocated in the instruction, while $\mathcal{M}_m(b) = 0$ for all remaining instructions. The cost expression *expr* is computed by applying the cost model of interest to all instructions of the rule, i.e., $expr = \mathcal{M}(g) + \sum_{i=1}^{n} \mathcal{M}(b_i)$.

Example 3 The application of the cost model $\mathcal{M}_i$ to the rule $while^c_{ol}$ returns 10 instructions as the rule contains this number of instructions (see the RBR on the left of Figure 2). ∎

Size relations. Given a program, COSTA performs a static analysis which captures, as much as possible, the effect of simple program statements on program variables. The information thus obtained is in the form of relations (linear constraints) among the values of variables at different program points. Such relations are referred to as *size relations* and relate 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 us, not only to bound the number of iterations of the loops, but also to compose the cost of different code fragments, since they allow translating equations over a set of variables into equations over another set of variables, such as the input parameters used to call a method. In particular, for each equation, we infer the size relations φ between the values of variables before the scope entry and the entry of its parent scope.

Example 4 Let us show the information captured by the size relations. We subscript the variables with 0 to refer to their initial values. For the rule *insertion_sort* the size relation $\{i = A - 2 \wedge A = A_0\}$ relates the initial values of the rule variables to the values before entering the loop at L4. Observe that it captures that A is not modified within this scope. For the rule $while_{ol}$, COSTA infers the size relation $\{j = i_0 + 1 \wedge A = A_0 \wedge i = i_0\}$ before calling the inner loop at L7, capturing the assignments of L5-L6. Additionally, for the rule $while^c_{ol}$, which corresponds to L11-L12, the size relation $\{i = i_0 - 1 \wedge A = A_0\}$ captures how i is modified in the loop. Analogously, for the inner loop (L8-L9) we have that $\{j_0 < A_0 \wedge j = j_0 + 1 \wedge A = A_0\}$ indicating that the value of j is incremented for each iteration of the loop. Condition $j_0 < A_0$ captures the condition of the guarded rule of the RBR. ∎

By applying the cost model to the instructions of each rule in the RBR and including the size relations, we obtain the CRS that corresponds to the program.

Example 5 We show the CRS for the running example in Figure 1:

$$
\begin{array}{lrll}
s_0 & insertion\_sort(A_0) & = c_1 + while_{ol}(A, i, 0, 0) & \{i = A_0 - 2, A_0 = A\} \\
s_{ol} & while_{ol}(A_0, i_0, j_0, v_0) & = c_2 & \{i_0 < 0\} \\
 & while_{ol}(A_0, i_0, j_0, v_0) & = c_3 + while_{il}(A, j) + & \\
 & & \quad while^{co}_{ol}(A, i_0, j', v) & \{i_0 \geq 0 \wedge j = i_0 + 1 \wedge A_0 = A\} \\
 & while^c_{ol}(A_0, i_0, j_0, v_0) & = c_4 + decrement(i_0) + & \\
 & & \quad while_{ol}(A, i, j_0, v_0) & \{i = i_0 - 1 \wedge A_0 = A\} \\
s_{il} & while_{il}(A_0, j_0, v_0) & = c_5 & \{j_0 \geq A_0\} \\
 & while_{il}(A_0, j_0, v_0) & = c_6 + while_{il}(A, j, v_0) & \{j_0 < A_0, j = j_0 + 1 \wedge A_0 = A\} \\
 & dec(x_0) & = c_{dec} & \{r = x_0 - 1\}
\end{array}
$$

It can be seen that each rule in the RBR (right of Figure 2) has its corresponding equation in the CRS. We will use c_i to represent the cost of applying a generic cost model $\mathcal{M}$ to the instructions of the corresponding rule. Note that we can easily define scopes s_0, s_{ol}, and s_{il} at the level of CRS as well, they are directly related to those of the RBR since equations originate from RBR rules. The expression c_1 corresponds to the cost of the bytecode instructions of the assignment at L3. Expressions c_2, c_3 and c_4 represent, respectively, the cost of evaluating the condition of the outer loop (L4), the cost of executing L5-L6 and the cost of L11-L12. Expressions c_5 and c_6 correspond, respectively, to the cost of evaluating the condition at L7 and to the cost of executing one iteration of the inner loop, i.e., L8 and L9. Similarly, c_{dec} represents the cost of executing the instructions of method `decrement`. The size relations of a CRS contain (1) the applicability conditions of the equations, e.g. $i_0 < 0$ and $i_0 \geq 0$ in the equations for $while_{ol}$ which are obtained from the guards, and, (2) the effect of simple program statements in the variables of the equation, e.g. $j = i_0 + 1$ in the equation $while_{ol}$ or $j = j_0 + 1$ in $while_{il}$. ∎

As we have seen, loops are represented by recursive equations in the CRS. Intuitively, in order to infer an UB for a single loop, we infer an UB *mexpr* on the worst-case cost of a single execution of its body and an UB *#iter* on the number of iterations that it can perform. Then, $\#iter * mexpr$ is an UB for the cost of the loop. To infer *mexpr* and *#iter*, we rely on three program analysis components: ranking functions, loop invariants and size relations, each described below.

Ranking functions. To bound the number of iterations performed by a given loop, we obtain in an automated fashion a linear function $\#iter$ from the loop variables to $\mathbb{N}$ which is strictly decreasing at each iteration. Such a function $\#iter$ is referred to as a *ranking function* in termination analysis. We use $\#iter_i(\overline{x})$ to denote the ranking function of the scope s_i in terms of the input arguments $\overline{x}$. The ranking functions produced by the cost analysis are of the form $\#iter(\overline{x}) = \mathsf{nat}(f(\overline{x}))$, where $\mathsf{nat}(f(\overline{x})) = \max(0, f(\overline{x}))$ ensures that ranking functions always return a non-negative value. In case the number of remaining iterations decreases logarithmically w.r.t. the value of the loop variables, we can also provide ranking functions of the form $\log(\mathsf{nat}(f) + 1)$, which are handled similarly.

Example 6 For the scope of the inner loop s_{il}, COSTA infers the ranking function $\#iter_{il}(A, j) = \mathsf{nat}(A - j)$ which safely bounds its number of iterations. For the outer loop s_{ol}, the ranking function inferred by COSTA is $\#iter_{ol}(i) = \mathsf{nat}(i + 1)$. ■

Loop invariants. For each recursive scope, using the size relations, we infer an invariant that involves the loop's variables and their initial values (i.e., their values before entering the loop). Each invariant ψ that involves loop variables $\overline{v}$ uses auxiliary variables $\overline{w}$ where w_i represents the initial value of v_i. The loop invariant is a disjunction between two sets of conditions $\psi = \psi^o \vee \psi^n$, the first one, ψ^o, corresponds to the first visit to the loop condition, and ψ^n corresponds to visiting the loop after executing the loop body at least once. Separating the invariant into these two cases will result in a more precise UB, and, in addition, it will help in verifying the invariant. These loop invariants, together with the size relations which are discussed below, are needed to compute the worst-case cost of executing one loop iteration.

Example 7 We use i_0 and j_0 to refer to the values of the variables `i` and `j` when entering the outer loop at L4. We use ψ_{ol} (ψ_{il}) to refer to the loop invariant of the outer (inner) loop. For this example, it can be easily seen that method `insertion_sort` does not modify the length of array `A`. Thus, for clarity of the presentation, in the loop invariants we do not show any constraint on the length of the array. However, COSTA infers and outputs this information. In the first visit to the loop condition, variables `i` and `j` are not modified with respect to their initial values, i.e.: $\psi^o_{ol} = \{i_0 = i \wedge j_0 = j\}$. Each iteration of the loop will decrement the value of variable `i` resulting in the constraint $\psi^n_{ol} = \{i \leq i_0\}$. Thus, the outer loop invariant becomes $\psi_{ol} = \psi^o_{ol} \vee \psi^n_{ol}$. For the inner loop, using j_0 and A_0 to refer to the value of `j` and `A` when entering the loop, we have that $\psi_{il} = \psi^o_{il} \vee \psi^n_{il} = (j_0 = j) \vee (j \geq j_0{+}1 \wedge A_0 \geq j)$. ■

Input-output size relations. Additional size relations in the computation of the UB are input-output size relations. These are linear constraints that relate the value returned by a given method to its input arguments. These input-output size relations are needed to propagate the effect of calling a procedure to its output variables. The techniques for UB inference are mostly modular, as CRS can be solved independently among them, and the results obtained for the different equations composed. However, a *global* size analysis that requires whole-program analysis sometimes is required in order to obtain more precise results on the cost. In the COSTA system, the use of global size analysis is optional.

Example 8 For method `int decrement(int x)` (L16) COSTA infers that the input argument `x` represented by x_0 and the value returned by `decrement`, denoted by r, have the size relation $r = x_0 - 1$. ∎

We can now put all pieces together: the following definition summarizes the computation of the UBs [3]. It starts by obtaining an UB for each *standalone* scope in the CRS which does not call any other, and consecutively replacing such UBs in the scopes which call them until all scopes are solved.

Definition 1 (Upper Bound [3]) Consider a CRS composed by equations of the form:

$$\langle C(\overline{x}) = expr + \sum_{i=1}^{k} Y_i(\overline{y}_i) + C(\overline{y}), \varphi \rangle \tag{1}$$

An upper bound for $C(\overline{x})$ is $UB_C(\overline{x}) = \#iter * mexpr$ such that:

1. $\#iter$ is the ranking function of the recursive calls of C,
2. the *invariant* ψ relates variables $\overline{y}_i \cup \overline{y}$ to their initial values $\overline{x}$,
3. UB_i are upper bounds for each call $Y_i(\overline{y}_i)$,
4. for each recursive equation, we have that:
   $mexpr'$=maximize($expr$,$\overline{x}$,ψ,φ)+ $\sum_{i=1}^{k}$ maximize($UB_i, \overline{x}, \psi, \varphi$)
   where function maximize($e, \overline{x}, \psi, \varphi$) returns the maximization of e for ψ and φ w.r.t. the equation entry variables $\overline{x}$,
5. if C has n recursive equations, $mexpr = max(mexpr'_1, ..., mexpr'_n)$, where $mexpr'_j$ is the maximized cost of equation j obtained in point 4, with $j = 1, \ldots, n$.

Let us illustrate Definition 1 on the running example:

Example 9 The process starts from the innermost (standalone) loop L7 of Figure 1 (s_{il}). The ranking function for the inner loop is $\#iter_{il}(A, j) = \mathsf{nat}(A - j)$. Executing the condition costs (at most) c_5 instructions. The cost of each iteration (i.e., the loop body) is c_6 instructions. Then, $UB_{il} = \mathsf{nat}(A_0 - j_0) * (c_5 + c_6) + c_5$ is an UB on the cost of this loop, where A_0 and j_0 are the values of `a` and `j` before entering in the inner loop.
Next, we proceed with the outer loop at L10. We use i_1 to refer to the value of i before entering the outer loop. Then, we have that $\#iter_{ol}(i_1) = \mathsf{nat}(i_1 + 1)$. The cost of the comparison is c_2 instructions. The code at L5–L6 costs c_3 instructions, and the code at L11–L12 (including the cost of the call to `decrement`) $c_4 + c_{dec}$ instructions. Then, the cost of each iteration of the outer loop is $c_2 + c_3 + \mathsf{nat}(A_0 - j_0) * (c_5 + c_6) + c_5 + c_4 + c_{dec}$, where the highlighted subexpression corresponds to UB_{il}, i.e., the cost of the inner loop computed above. Note that in this case, each iteration of the inner loop might have a different cost, since $A_0 - j_0$ is not the same for all iterations. Simply multiplying the number of iterations $\mathsf{nat}(i_1 + 1)$ by such a cost is unsound. The solution is to find an expression U in terms of the initial values of A, i, j in the outer loop (A_1, i_1, j_1) which does not change during the loop such that $U \geq A_0 - j_0$ in all iterations. Note that constant expressions do not need to be maximized. Then, $UB_{ol} = \mathsf{nat}(i_1 + 1) * [c_2 + c_3 + \mathsf{nat}(U) * (c_5 + c_6) + c_5 + c_4 + c_{dec}] + c_2$ is an UB for the outer loop. In order to find such a U, COSTA uses the loop invariant ψ_{ol} (see Example 7) and the

size relations of the equation $while_{ol}$ (see Example 4) by means of the function $maximize(A_0 - j_0, \langle A_1, i_1, j_1\rangle, \psi_{ol}, \varphi_{while_{ol}})$, which solves the parametric integer programming (PIP) problem of maximizing the objective function $A_0 - j_0$ w.r.t. $\psi_{ol}, \varphi_{while_{ol}}$ where A_1, i_1, j_1 are the input parameters. This produces an expression in terms of A_1, i_1, j_1 which is greater than or equal to $A_0 - j_0$ in all iterations of the loop. In our example, this maximized expression is $U = A_1 - 1$.
We finally can compute the cost of the `insertion_sort` method. Let us assume that the cost of L4 is c_1, then the cost of the whole method is

$$c_1 + \mathsf{nat}(i_1 + 1) * [c_2 + c_3 + \mathsf{nat}(A_1 - 1) * (c_5 + c_6) + c_5 + c_4 + c_{dec}] + c_2.$$

We need to express this UB in terms of the input parameter `A.length`. For this, COSTA maximizes (using PIP) the expressions $i_1 + 1$ and $A_1 - 1$ w.r.t. the size relations of the equation *insertion_sort*, i.e., $\{i = A_0 - 2, A = A_0\}$, and obtains $A - 1$ for both expressions. Therefore, the UB for `insertion_sort(A)` is

$$c_1 + \mathsf{nat}(A_0 - 1) * [c_2 + c_3 + \mathsf{nat}(A_0 - 1) * (c_5 + c_6) + c_5 + c_4 + c_{dec}] + c_2. \quad \blacksquare$$

2.2 From COSTA to the Annotated Java Program

In the verification phase it suffices to prove the correctness of the inferred ranking function, loop invariants, and size relations: based on these, it is straightforward to compute an UB for the method by applying PIP to obtain *mexpr* and then simply multiply $\#iter * mexpr$. Furthermore, the cost executed by each instruction is obtained by applying the cost model that only maps each bytecode instruction to its corresponding cost. Correctness of this cost model is trivial as it is a simple mapping from each instruction to a number. . In this section we will describe the different elements generated by COSTA to compute the UB and their translation into JML annotations on the Java program, which allows their verification.

As COSTA works at the level of bytecode and KeY uses an annotated Java file we have to relate the bytecode program to the source program. To relate the RBR program to the Java source code, COSTA retains the variable identifiers of the form l_i, used by the bytecode program. To relate the bytecode instructions to their position in the Java program, COSTA stores in the RBR the program counter of all bytecode instructions. With this information in the RBR, we use the debugging information available in the `.class` file to relate the bytecode instructions to their corresponding variable names and source code line numbers in the Java program. By means of this debugging information, COSTA is able, not only to output in the JML annotations the source code variable names, but also is able to output the annotations in the proper place of the Java program. We use the following structures from the `.class` file ([27]): (1) the `LocalVariableTable` attribute, which relates each bytecode variable with its corresponding Java source variable name; and, (2) the `LineNumberTable`, which relates the program counter of the bytecode instructions, with their corresponding source code line number. With this information we can not only generate the JML annotations with the source code variable names, but also automatically determine the source code line where a scope starts (finishes), i.e., the line of the first (last) instruction of the scope to print the annotations.

Example 10 The `LineNumberTable` for method `insertion_sort` of the Java program of Figure 1 contains the following information:

`line 3: bc 0`	`line 6: bc 12`	`line 9: bc 28`	`line 14: bc 52`
`line 4: bc 5`	`line 7: bc 17`	`line 7: bc 31`	`line 4: bc 58`
`line 5: bc 8`	`line 8: bc 20`	`line 13: bc 45`	`line 14: bc 62`

By using the first instruction of the rule $while_{ol}$ which has program counter `58` (see the RBR on the left of Figure 2) and by means of the entry `line 4: bc 58` we can determine that the loop condition is placed at L4 of the source code. Analogously, we can determine that the last instruction of the outer loop, that is `57: istore_2`, is placed at L12: as $52 \leq 57 < 58$, we use the entry `line 14: 52`. This information is used to output the JML annotations for each scope.
We also use the local variables of `insertion_sort` indicating that l_0 corresponds to `this`, l_1 to `A`, and l_2, l_3 and l_4 to `i`, `j` and `v`, respectively. With this information we can translate the variables used in the rule $while^c_{ol}(\langle l_0, l_1, l_2, l_3, l_4\rangle, \langle l_2, l_3, l_4\rangle)$ (on the right of Figure 2) to $while^c_{ol}(\langle this, A, i, j, v\rangle, \langle i, j, v\rangle)$. This translation is used during generation of JML annotations as described below. As arrays are abstracted to their length, instead of A we use $A.length$ in the annotations. ■

Ranking functions. For each recursive construct in the CRS, which corresponds to a loop in the source code, COSTA computes a ranking function. As we have seen in Section 2.1, the ranking function is wrapped by the `nat` function, which is an `if-then-else` structure that serves to keep the ranking function non-negative. Thus, we translate $\#iter = \mathsf{nat}(exp)$ to the JML annotation

$$//@decreasing\ exp > 0\ ?\ exp : 0$$

which encodes $\max(0, exp)$. According to the JML specification, such an annotation must be added above the loop it refers to.

Example 11 Line 17 of Figure 3 shows the JML annotation generated by COSTA for the ranking function $\#iter_{il}$ where A abstracts the length of the array (shown in Example 6):

$$//@decreasing\ A.length{-}j > 0\ ?A.length{-}j\ :0$$

Similarly, for the outer loop ranking function, $\#iter_{ol}$, COSTA outputs at L9:

$$//@decreasing\ i{+}1 > 0\ ?\ i{+}1\ :\ 0.$$

■

Loop invariants. Loop invariants relate the loop's variables and their initial values. For this reason it is necessary to declare specification variables, known as *ghost* variables, to capture the variable values before entering a loop. Thus, the JML annotations for invariants each consist of one line defining all variables involved in the loop invariant $\bar{w}$ as ghost variables

$$//@\ ghost\ int\ w_1 = v_1; \ldots; int\ w_n = v_n$$

and an annotation for declaring the loop invariant itself, inserted above the loop it refers to:

$$//@\ loop\_invariant\ \psi$$

```
1 void insertion_sort(int A[]) {
2     int i, j, v;
3     //@ ghost int i0 = i; int j0 = j; int A0 = A.length; int v0 = v;
4      i=A.length-2;
5     //@ assert (i = A0−2 ∧ A.length = A0);
6     //@ ghost int i1 = i; int j1 = j; int A1 = A.length;
7     //@ ghost int i2 = i; int j2 = j;
8     //@ loop_invariant (i2 = i ∧ j2 = j) ∨ (i ≤ i2)
9     //@ decreasing i + 1>0 ? i + 1 : 0;
10    while ( i>=0 ) {
11        j=i+1;
12        v=A[i];
13        //@ assert i0 ≥ 0 ∧ j = i1 + 1 ∧ A1=A.length;
14        //@ ghost int j3 = j; int A3 = A.length;
15        //@ ghost int j4 = j; int A4 = A.length;
16        //@ loop_invariant (j4 = j) ∨ (j ≥ j4+1 ∧ A4 ≥ j)
17        //@ decreasing A.length − j>0 ? A.length − j : 0;
18        while ( j<A.length && A[j]<v) {
19            A[j-1]=A[j];
20            j++;
21            //@ assert (j3 < A3 ∧ j = j3 + 1 ∧ A3 = A.length);
22            //@ set j3 = j; A3 = A.length;
23        }
24        A[j-1]=v;
25        i = decrement(i);
26        //@ assert (i = i1 − 1 ∧ A1 = A.length);
27        //@ set i1 = i, j1 = j, A1 = A.length;
28    }
29 }
30 /*@ public behavior
31   @ requires true;
32   @ ensures \result = x −1;
33   @ signals_only Exception;
34   @ signals (Exception) true;
35   @*/
36 int decrement (int x) {
37     return x - 1;
38 }
```

Fig. 3 Java source code for the running example working on Integer data

Observe that ghost variables take the values of the program variables at the annotation program point, in our case, before entering the loop.

Example 12 As we need unique identifiers for the ghost variables created for the different scopes of the program, we subscript the ghost variables with numbers. For the invariant of the outer loop, we first declare ghost variables i_2, j_2 at L7 of Figure 3. To capture their initial value before entering the loop, they are initialized with the values that variables `i`, `j` have at this program point. As the loop invariant

computed in Example 7 for the outer loop is composed of the disjunction of two sets of conditions $\psi^o_{ol} \vee \psi^n_{ol}$, at L8 we have the annotation:

$$//@\ loop\_invariant\ (i_2 = i \wedge j_2 = j) \vee (i \leq i_2).$$

Similarly, for the inner loop (L18), ghost variables j_4, A_4 are declared and initialized at L15, and the loop invariant captures that the value of `j` is larger than its value in the first iteration but that it does not exceed the length of the array (L16 of Figure 3):

$$//@\ loop\_invariant\ (j_4 = j) \vee (j \geq j_4 + 1 \wedge A_4 \geq j).$$

■

Size relations. 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 $\bar{w}$ their values at line L_p. To capture the values at the beginning of the scope, 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 that program point. These annotations suffice for non-recursive scopes. However, for loops, as variables might be modified at each iteration, at the end of the loop we have to update the values of the ghost variables, by means of the annotation *//@ set* $w_1 = v_1$*;...;* $w_n = v_n$;, which updates the values of the ghost variables to the values of the variables at the end of the scope.

Example 13 We start with the annotations for the scope of equation *insertion_sort*, i.e., the assignment at L4 of Figure 3. We first output the annotation at L3 declaring the ghost variables j_0 and A_0, assigning the values of `j` and `A.length` to ghost variables. Then, at the end of the scope s_0, i.e., immediately before the outer loop (L5) we add an annotation to output the size relations of the equation *insertion_sort* (see Example 5):

$$//@\ assert\ (i = A_0 - 2 \wedge A.length = A_0).$$

As ghost variables for different scopes must have different identifiers, we declare ghost variables i_1, j_1 and A_1 (L6) for the size relations of the outer loop. Observe that $\varphi_{while_{ol}}$, which corresponds to the size relations of L11-L12 are added at L13 and $\varphi_{while^c_{ol}}$, which corresponds to L24-L25, are added at L26. Since these code fragments appear inside a loop, the values of i_1, j_1 and A_1 should be updated at each iteration. This is done by modifying them at each iteration at the end of the outer loop (L27). The size relation at L26, which states that `i` is increasing, is used by COSTA to synthesize the ranking function. This also helps KeY in proving that it is indeed a ranking function. ■

Input-output size relations. These are needed to propagate the effect of calling a procedure to its output variables. Input-output size relations are provided by means of a JML *method contract*.

/@ public behavior*
@ requires true;
@ ensures φ_{io}*;*

```
@ signals_only Exception;
@ signals (Exception) true;
@*/
```

Example 14 COSTA outputs the annotation *//@ensures \result=x−1* in the contract for `decrement`, which is shown in lines 30–34 of Figure 3, and it can be used to verify the number of iterations of the outer loop because `decrement` is called at L25. ■

Figure 4 shows the algorithm that inserts the annotations inferred by COSTA in the Java source code. The algorithm first traverses the scopes of the RBR program (in the first loop), and for each one it creates corresponding annotations in the Java program, then it traverses the methods of the RBR program, and also generates corresponding annotations in the Java program. It is important to note that in the algorithm, scope s refers to an RBR scope, however, since scopes at the level of the RBR and the CRS are directly related, we refer to the equations of s for referring to the corresponding equations in the CRS. Note also that recursive scopes in the RBR correspond loops in the Java program (since we have assumed that the Java program is recursion-free). Next we explain the algorithm in details.

Functions $first\_line(s)$ and $last\_line(s)$ ($last\_line(e)$) refer to the first and the last line of the source code for scope s (or CRS equation e). We use $before(s)$ ($before(L)$) to refer to the line of the source immediately before scope s (line L). We use $decreasing(s)$, $loop\_invariant(s)$ and $set(s)$, $ghost\_inv(s)$, $ghost\_size(s)$, $assert(s)$ to refer to the JML annotations for scope s. In addition we use $assert(eq)$ to refer to the JML annotation created by using the size relations of eq, i.e. φ_{eq}. Function $print(L, annotation)$ writes $annotation$ at line L in the Java source code. We also use $contract(M)$ to refer to the JML method contract for method M. In the algorithm we see that, for recursive scopes, i.e. loops, we print the annotations *decreasing*, *loop_invariant*, *ghost_inv*, *ghost_size*, immediately before the loop itself, while for non-recursive scopes we print the *ghost_inv* in the first line of the scope. In recursive scopes we print the *assert* annotations in the last line of those equations that contain calls to other equations, while *assert* is printed at the end of the non-recursive scopes.

Example 15 In the program of Figure 3, we have two recursive scopes s_{il}, s_{ol}, and one non-recursive scope s_0. The annotations at L8, L9 correspond to the JML annotations for the loop invariant and the ranking function of s_{ol} and are added before the outer loop. In addition, annotations at L6, L13, L26 and L27 provide the size relations of s_{ol}. Observe that the annotation at L13 corresponds to the size relations of the equation $while_{ol}$, whereas the size relations of $while^c_{ol}$ are printed in the last line of the iterative scope s_{ol} (L26). Note that both annotations use the ghost variables declared at L6. Similarly, annotations at L14, L15, L16, L17, L21 and L22 correspond to the JML annotations for s_{il}. The size relations of s_0 only contain the annotations at L3 to define the ghost variables and L5 to annotate them. In addition, the algorithm includes the method contract of `decrement` immediately before its declaration at L30. In the example, we do not include the `insertion_sort` contract because it does not contain any relevant information in the *//@ensures* clause. ■

for all $s \in Program$ **do**
  if s is recursive **then**
    $print(before(s), ghost\_size(s))$
    $print(before(s), ghost\_inv(s))$
    $print(before(s), loop\_invariant(s))$
    $print(before(s), decreasing(s))$
    $print(last\_line(s), set(s))$
    for all $eq \in s$ **do**
      if (eq contains calls) **then**
        $print(last\_line(eq), assert(eq))$
      end if
    end for
  else if s is non-recursive **then**
    $print(first\_line(s), ghost\_inv(s))$
    $print(last\_line(s), assert(s))$
  end if
end for
for all $M \in Program$ **do**
  $print(before(M), contract(M))$
end for

Fig. 4 Algorithm to output the JML annotations in the Java source code

2.3 Verification of Upper Bounds

We now describe the verification techniques used in KeY to prove program correctness, focusing on those relevant to UB verification.

2.3.1 Verification by Symbolic Execution

The program logic used by KeY is *JavaCard Dynamic Logic* (JavaDL) [15], a first-order dynamic logic with arithmetic. Programs are first-class citizens similar to in Hoare logic, but in dynamic logic correctness assertions can appear arbitrarily nested. JavaDL extends sorted first-order logic by a program modality $\langle\cdot\rangle\cdot$ (read "diamond"). Let p denote a sequence of executable Java statements and ϕ an arbitrary JavaDL formula, then $\langle\mathsf{p}\rangle\phi$ is a JavaDL formula which states that program p terminates, and in its final state ϕ holds. A typical formula in JavaDL looks like

$$\mathsf{i} \doteq i_0 \wedge \mathsf{j} \doteq j_0 \rightarrow \langle\overbrace{\mathsf{i=j-i;j=j-i;i=i+j;}}^{\mathsf{p}}\rangle(\mathsf{i} \doteq j_0 \wedge \mathsf{j} \doteq i_0)$$

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 i_0, j_0 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 i_0, j_0, 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:

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

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:

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

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 location (program variable, field, array) and a value. The meaning of updates is the same as that of an assignment, but they can be composed in different 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. Application of updates is similar to explicit substitutions, but is aware of aliasing.

Loops and recursive method calls give rise to infinitely long symbolic executions. Invariants are used to deal with unbounded program structures (an example is given below). Exhaustive application of symbolic execution and invariant rules results in formulas of the form $\{u\}\langle\rangle\phi$ where the program in the modality has been fully executed. At this stage, symbolic updates are applied to the postcondition ϕ resulting in a first-order formula that represents the weakest precondition of the executed program w.r.t. ϕ. In case of the above swap example

$$\Longrightarrow \mathsf{i} \doteq i_0 \wedge \mathsf{j} \doteq j_0 \rightarrow \langle \overbrace{\mathsf{i=j-i;j=j-i;i=i+j;}}^{\mathsf{p}} \rangle (\mathsf{i} \doteq j_0 \wedge \mathsf{j} \doteq i_0)$$

we execute the program p symbolically using the assignment rule. After finishing with symbolic execution and interleaved application of other first-order sequent rules, we end up with the following sequent

$$\mathsf{i} \doteq i_0 \wedge \mathsf{j} \doteq j_0 \Longrightarrow \{\mathsf{i} := \mathsf{j} \parallel \mathsf{j} := \mathsf{i}\} (\mathsf{i} \doteq j_0 \wedge \mathsf{j} \doteq i_0)$$

we can now compute the weakest precondition of the postcondition by applying the update. The resulting sequent

$$\mathsf{i} \doteq i_0 \wedge \mathsf{j} \doteq j_0 \Longrightarrow (\mathsf{j} \doteq j_0 \wedge \mathsf{i} \doteq i_0) \ .$$

is trivially valid.

2.3.2 Proof-Obligation for Verifying Upper Bounds

To verify UBs in KeY the annotated source code files provided by COSTA are loaded. For methods where COSTA did not generate a contract, KeY provides the following default contract:

```
/*@ public behavior
  @ requires  true;
  @ ensures  true;
  @ signals_only  Exception;
  @ signals  (Exception) true;
  @*/
```

This contract requires to prove termination for any input and ensures that all possible execution paths are analyzed. Abrupt termination by uncaught exceptions is allowed (signals clauses). To prove that a method m satisfies its contract, a JavaDL formula is constructed which is valid iff m satisfies its contract. Slightly simplified, for insertion_sort the JavaDL proof obligation formula (using the default contract) is:

$$\forall o; \forall a_0; \{\mathsf{a} := a_0 \parallel \mathsf{self} := o\}(\neg(\mathsf{a} \doteq \mathsf{null}) \land \neg(\mathsf{self} \doteq \mathsf{null}) \rightarrow \langle \mathsf{try}\ \{\ \mathsf{self.insertion\_sort(a)@NestedLoops};\ \}\ \mathsf{catch(Exception\ e)\{\ exc{=}e;\ \}}\rangle(\ \mathsf{exc} \doteq \mathsf{null} \lor \mathrm{instance}_{\mathsf{Exception}}(\mathsf{exc}))$$

The above formula states that for any value o of self and any value $a0$ of the argument a which satisfies the implicit JML preconditions (self and a are not null), the invocation of method self.insertion_sort(a) declared in class NestedLoops *terminates* (required by the use of the diamond modality) and in its final state either no exception has been thrown or any thrown exception was of type Exception.

2.3.3 Verification of Proof-Obligations

The proof obligation formula must be proven valid by executing the method insertion_sort symbolically starting with the execution of the variable declarations. Ghost variable declarations and assignments to ghost variables (*//@ set var=val;*) are symbolically executed just like Java assignments.

Verifying Size Relations. 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 when ϕ holds. In the insertion_sort example, a proof split occurs exactly before entering each loop. This verifies the size relations among variables as derived by COSTA and encoded in terms of JML assertion statements (see Section 2.1). Input-output size relations encoded in terms of method contracts are proven correct as outlined in Section 2.3.2.

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. The loop invariant rule is closely related to the standard Hoare loop invariant rule for total correctness, but

provides an optimized handling for framing (similar to a partial havoc as realized in [14]):

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

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 while remaining non-negative. Premise (iii) continues symbolic execution upon loop exit. The integer-typed variant term ensures loop termination as it has lower bound 0 and is decreased in 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.

Generated Proofs. A single proof for each method is sufficient to verify the correctness of the derived loop invariants, ranking functions and size relations. The reason is that the contracts capturing the input-output size relations are not more restrictive w.r.t. the precondition than the default contracts. Hence, with the verification of the input-output size relation contracts, we analyze all feasible execution paths and prove correctness of all loop invariants, ranking functions and JML assertion annotations as part of proving that the method satisfies its contract.

We stress that the proofs run fully automatic. Much of the time is needed to derive specific instances of arithmetic properties. As future work, we plan to do proof profiling and to reduce the search time by hashing frequently occurring normalisation steps.

3 Verified Upper Bounds for Heap Manipulating Programs

In the previous section, we studied formal verification of inferred resource guarantees that depend only on integer data passed as method parameters. In realistic programs, however, resource consumption is often bounded by the size of heap-allocated data structures. Bounding their size requires a number of structural heap analyses. In this section, we first identify what exactly needs to be verified to guarantee a sound analysis of heap manipulating programs, and then we provide a suitable extension of the program logic used for verification to handle structural heap properties in the context of resource guarantees.

3.1 Resource Analysis 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

```
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 x0 = x;
6     List p = null;
7     List c = x;
8     List n = new List(v, null);
9     //@ ghost List c0 = c
10    //@ assert \depth(n) = 1 ∧ \depth(c0) = \depth(x0)
11    //@ decreasing \depth(c)
12    //@ loop_invariant \depth(c0) ≥ \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 ∧ x.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. 5 The running example, with (partial) JML annotations

terms of some measure on the size 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 16 Consider the program in Figure 5 where class `List` implements a linked list as usual. For method `insert`, COSTA infers the UB $c_1 * \mathtt{nat}(x) + c_2$ where x refers to the length of `x` and c_1, c_2 are constants representing the cost of the instructions inside, before and after the loop. The UB depends on the length of `x`, because the list is traversed in L16–L19. ∎

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 a node of the structure to a leaf node. Using this size

measure, COSTA infers size relations, ranking functions and loop invariants 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 Section 2. In the following, we identify the essential information of the path-length analysis (and related analyses) required by KeY.

3.1.1 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. To verify the path-length information with KeY, we extended JML with the new keyword `\depth` that gives the depth of a data structure to which a reference variable points. In particular, for ranking functions, invariants, size-relations, and contracts, if the corresponding constraints include a variable x, related to a reference variable `x`, we replace all occurrences of x by `\depth(x)`.

Example 17 We explain the various path-length relations inferred by COSTA for the method `insert` of Figure 5, and how they are used to infer an UB. We only show the annotations of interest. For the loop at L16–L19, COSTA infers that the depth of the data structure to which `c` points decreases at each iteration. Since the depth is bounded by 0, it concludes that $\mathsf{nat}(c)$ is a ranking function for that loop. As 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 the depth of `x` we have that $c_1 * \mathsf{nat}(x) + c_2$ 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 L3) 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 variable certainly points to an acyclic data structure; and (b) which variables might share common regions in the heap.

3.1.2 Cyclicity analysis

A cyclicity analysis infers information about 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 the abstract state before the instruction 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 extended 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 L13 of the program in Figure 5) and for contracts as pre- and postconditions (as in L1 and L2 of Figure 5). To make use of the reachability relations we extended JML by the new keyword $\texttt{\textbackslash reachPlus}(\texttt{x},\texttt{y})$, which indicates that `y` *must* be reachable from `x` in at least one step, and use the standard keyword $\texttt{\textbackslash reach}(\texttt{x},\texttt{y})$ 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 $!\texttt{\textbackslash reachPlus}(\texttt{x},\texttt{y})$ whenever $\texttt{x}\rightsquigarrow\texttt{y} \notin A$, and we add $!\texttt{\textbackslash reach}(\texttt{x},\texttt{y})$ whenever $\texttt{x}\rightsquigarrow\texttt{y} \notin A \wedge \texttt{x}\diamond\texttt{y} \notin A$ (for example, in L15 of Figure 5).

Example 18 To include the acyclicity information in the JML annotations, COSTA adds to the *//@loop_invariant* the results of the acyclicity analysis, as it can be seen at L13 of Figure 5. In addition, COSTA includes in the method contract *//@requires* annotation that the received list must be acyclic (L1), and that the list returned by `insert` remains acyclic in the *//@ensures* clause (L2). Moreover, to include the reachability information to the annotations, COSTA extends the loop invariants of the program (for example, in L15). ■

3.1.3 Sharing analysis

Knowledge about possible sharing is required by both path-length and cyclicity analyses. 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 something 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.

Example 19 Sharing information is shown at L14 of Figure 5, where COSTA adds to the invariant the `disjoint` information, which annotates that the data structure referenced by `n` does not share with `x`, `c`, `p`. ■

3.2 Verification of Structural Heap Properties

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 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 article and implemented in KeY.

3.2.1 Heap Representation

First we briefly explain the logical modeling of the heap in JavaCard DL[2]. The heap of a Java program is represented as an element of type $Heap$. The $Heap$ 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 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 Section 2.3). It simply updates the global heap program variable with the updated heap object:

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

3.2.2 Predicates for Structural Heap Properties

For the sake of readability, in Section 3.1, we gave simplified versions of the predicates `\depth`, `\acyclic`, `\reach`, `\reachPlus` and `\disjoint` as compared to their actual implementation. In reality, these predicates have an extra argument that restricts their domain to a given set of fields. For example, instead of `\depth`(x) we might have actually `\depth`$(\{x.\texttt{next}\}, x)$ which refers to the depth of `x` considering

[2] Note that this is *not* the heap model described in earlier publications on KeY such as [15]. In the present paper we use an explicit heap model based on [36] which is implemented in KeY starting with version 2.0.

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.

Finally, 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

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

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 $n \geq 0 \wedge s \neq \mathsf{null} \wedge ((n \doteq 0 \wedge s \doteq e) \vee \exists f.(o, f) \in l \wedge \texttt{\textbackslash reach}(h, l, o.f, e, n-1))$. 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 [36] for a full account). Here we need only three location set constructors: the constructor *empty* for the empty set, the constructor $\mathit{singleton}(o, f)$ 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 $\mathit{allObjects}(f)$ which stands for the location set $\{(o, f) \mid o \in \mathit{Object}\}$.

Example 20 $\texttt{\textbackslash reach}(h, \mathit{allObjects}(\mathsf{next}), \mathit{head}, \mathit{last}, 5)$ 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 have directly axiomatized structural heap predicates such as $\texttt{\textbackslash acyclic}(h, l, o)$ or $\texttt{\textbackslash disjoint}(h, l, o, u)$. Instead we preferred 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 Section 3.1.2 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 $\texttt{\textbackslash reachPlus}(h, l, o, u)$ 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 definition of predicate `\reachPlus` is directly based on the definition of `\reach`. This definition can be directly used by the calculus using the rewrite rule:

$$\texttt{\textbackslash reachPlus}(h, l, o, u) \rightsquigarrow \exists n.(n > 0 \wedge \texttt{\textbackslash reach}(h, l, o, u, n))$$

However, the application of the above rewrite rule is delayed as long as possible and instead auxiliary lemmas are used. 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.

The structural heap predicate `\acyclic` is axiomatized as follows

$$\begin{aligned}&\texttt{\textbackslash acyclic}(h, l, o) \rightsquigarrow \\ &\quad \neg\exists ce; (\texttt{\textbackslash reachPlus}(h, l, o, ce) \wedge \texttt{\textbackslash reachPlus}(h, l, ce, ce))\end{aligned}$$

A brief justification that the above definition captures the intended meaning: Assume the data structure rooted at object o has a cycle, i.e., there is a sequence

$(o =) u_0 \cdot u_1 \cdot \ldots \cdot u_n (= u_i)$. Hence, there is a node $ce = u_i$ (e.g., u_1) such that u_i is reachable from o in more than one step and u_i can be reached from u_i by one or more steps. Note this definition holds also for cycles directly rooted at object ce.

Finally, we provide the definition of $\backslash\texttt{disjoint}(h, l, o, u)$ which expresses that the two linked (w.r.t. the locations contained in l) data structures rooted in o and u do not share any common object. Its axiomatisation is as follows:

$$\backslash\texttt{disjoint}(h, l, o, u) \rightsquigarrow (\neg(o \doteq u) \vee o \doteq \mathsf{null}) \wedge \neg\exists c; (\backslash\texttt{reachPlus}(h, l, o, c) \wedge \backslash\texttt{reachPlus}(h, l, u, c))$$

The provided definitions are only expanded as a last resort. Most of the times lemmas are used instead. To ensure the correctness of lemma rules derived from axioms/definitions, KeY provides the facility to generate proof obligations for rules [19] and to prove those correct within the system.

The following section describes one central difficulty that arises when reasoning about structural heap properties and how we solved it to achieve higher automation.

3.2.3 Field Update Independence

When reasoning about structural heap predicates one often ends up in a situation where we have 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 $\backslash\texttt{acyclic}(h, l, u)$ holds and have to prove that after executing the assignment `o.f=v;` the formula $\backslash\texttt{acyclic}(\mathit{store}(h, o, f, v), l, u)$ holds. The lemma used to eliminate such dependencies is realized as the following rewrite rule

$$\begin{array}{l}\backslash\texttt{acyclic}(\mathit{store}(h, o, f, v), l, u) \rightsquigarrow \\ \quad \texttt{if}(\neg(\{(o, f)\} \subseteq l)) \\ \quad\quad \texttt{then}\ (\backslash\texttt{acyclic}(h, l, u)) \\ \quad\quad \texttt{else}\ (\backslash\texttt{acyclic}(\mathit{store}(h, o, f, v), l, u))\end{array}$$

The rule replaces the acyclic expression by a conditional formula, which evaluates to the term specified in the `then` part, if the condition $\neg(\{(o, f)\} \in l)$ evaluates to true, i.e., the store location is not contained in the set of locations on which the acyclic predicate depends. Otherwise, it evaluates to the original formula. The proof search strategies apply this rule only, if they can determine that the condition might become true. Particular care is taken to avoid repeated (useless) applications on the same occurrence of the acyclic formula. 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 situations, 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 $\backslash\texttt{acyclic}(\mathit{store}(h, \mathsf{o}, \mathsf{data}, 5), l, u)$ to $\backslash\texttt{acyclic}(h, l, u)$.

2. Checking whether an object o whose field has been updated is reachable from one of the objects in its context is more expensive than the previous case, but still cheaper than a full analysis. For example, we can check whether the object o is reachable from object u in case of $\texttt{\textbackslash acyclic}(store(h, o, f, v), l, u)$. If the answer is negative we can again discard the store expression.

3.2.4 Path-Length Axiomatization

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

$$\text{depthDef}\ \frac{\begin{array}{l} skDepth \doteq \texttt{\textbackslash depth}(h, l, o) \land \texttt{\textbackslash acyclic}(h, l, o) \rightarrow (\\ \quad ((o \doteq \mathsf{null} \land skDepth \doteq 0) \lor \\ \quad\quad \exists u.(\texttt{\textbackslash reach}(h, l, o, u, skDepth - 1)) \land \\ \quad\quad \forall u.(\forall dist.((\texttt{\textbackslash reach}(h, l, o, u, dist) \rightarrow dist \leq skDepth - 1))))) \\ \Longrightarrow \\ \quad \phi[skDepth] \end{array}}{\Gamma \Longrightarrow \phi[\texttt{\textbackslash depth}(h, l, o)], \Delta}$$

where *skDepth* is a fresh skolem constant and u, *dist* do not occur freely in h, l, o. A similar definition exists for $\texttt{\textbackslash depth}$ occurring on the left side of the sequent. 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 $\texttt{\textbackslash depth}(store(h, o, f, v), l, u)$ there is the lemma below 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 $\texttt{\textbackslash depth}$ in case of a heap update as detailed in Section 3.1.1:

$$\text{depthFieldStore}\ \frac{\begin{array}{l} \neg o \doteq \mathsf{null} \land \\ \quad (\texttt{\textbackslash acyclic}(store(h, u, f, v), l, u) \land \{(u, f)\} \subseteq l \rightarrow \\ \quad (\forall z.((\texttt{\textbackslash acyclic}(store(h, u, f, v), l, z) \land \\ \quad\quad (\texttt{\textbackslash reachPlus}(store(h, u, f, v), l, z, u) \lor z \doteq u)) \rightarrow \\ \quad\quad (\texttt{\textbackslash depth}(store(h, u, f, v), l, z) \leq \texttt{\textbackslash depth}(h, l, z) + \texttt{\textbackslash depth}(h, l, v) \land \\ \quad\quad\ \texttt{\textbackslash depth}(store(h, u, f, v), l, z) > \texttt{\textbackslash depth}(h, l, v)))) \\ \Longrightarrow \\ \phi[\texttt{\textbackslash depth}(store(h, u, f, v), l, o)]) \end{array}}{\Gamma \Longrightarrow \phi[\texttt{\textbackslash depth}(store(h, u, f, v), l, o)], \Delta}$$

4 Heap-Sensitive Verification of Upper Bounds

In this section, we focus on the verification of upper bounds that require *field-sensitive* analysis [4]. This happens when the resource consumption depends on

shared mutable data stored in class fields. Shared and mutable data pose major problems in static analysis as well as in verification and many systems are unable to keep track of the values of class fields and array elements. We describe how we perform field-sensitive resource analysis and its verification.

4.1 Heap-Sensitive Resource Analysis

Field-sensitive static analysis is challenging, because (1) there can be multiple references to the same memory location, and (2) because a memory location can be accessed using a reference variable whose value can change during the execution (and can be used to point to a different memory location). The approach in [4] is based on the idea that, when fields behave like local variables, they can be transformed into local integer variables and use a field-insensitive analysis to infer information on them. With "behaving like local variable" we mean that (1) there are no multiple aliases to it, and (2) the reference variable through which the field is accessed does not change its value. Let us look at some examples:

```
while (x.f>0){
   x.f--;
}
```

```
while (x.f>0){
   x.f--;
   y.f++;
}
```

```
while (x.f>0){
   x.f--;
   aux=y.f;
}
```

```
while (x.f>0){
   x.f--;
   x=x.next;
}
```

In the leftmost loop both restrictions mentioned above do hold and COSTA is able to infer that `nat(x.f)` is a ranking function for the loop. In the second loop condition (1) does not hold because we might have multiple aliases to the field `f` through `x` and `y`. However, the paper [2] proposes an improvement for numeric fields (it would not be correct for reference fields) that allows tracking the field `f` if the write access is always through the same reference. This enhancement allows to infer a resource bound for the third loop, because the field `f` is only written to through the reference variable `x`, even though it is read also from `y`. COSTA is able to infer that $\mathsf{nat}(x.f)$ is a ranking function for that loop. The rightmost loop shows an example for which condition (2) does not hold, because the value of the reference `x` changes at each iteration.

Arrays are treated in a similar manner. In the next example for the program on the left the array element is addressed locally and both conditions hold, while in the example on the right the reference variable changes and thus we cannot consider the array element as local.

```
while (a[i]>0){
   a[i]--;
}
```

```
while (a[i]>0){
   a = b;
   a[i]--;
}
```

Whenever COSTA is able to prove the locality conditions on fields and arrays, it simply infers annotations as seen in Sections 2, 3 using the involved fields.

Example 21 The following example shows the annotated source code generated by COSTA applying the heap-sensitive analysis for handling fields (left) and array elements (right).

```
1 //@ghost int g = x.f;
2 //@loop_invariant (g = x.f)∨(g ≥ x.f+1);
3 //@decreasing x.f ≥ 0 ? x.f : 0);
4 while (x.f>0){
5   x.f--;
6   //@assert x.f = g−1
7   //@set g = x.f
8 }
```

```
1 //@ghost int g = a[i];
2 //@loop_invariant (g = a[i])∨(g ≥ a[i]+1);
3 //@decreasing a[i] ≥ 0 ? a[i] : 0);
4 while (a[i]>0){
5   a[i]--;
6   //@assert a[i] = g−1
7   //@set g = a[i]
8 }
```

It can be seen that array and field accesses are handled in a uniform manner by COSTA. At L1 of both programs ghost variables are created, L2 contains the loop invariant and L3 the ranking function. L6 and L7 contain the size relation annotations. ■

4.1.1 Aliasing Preconditions

An interesting aspect of the field-sensitive analysis in COSTA is that it is able to infer aliasing preconditions under which upper bounds can be found. Let us consider two further examples:

```
while (x.f>0){
   x.f--;
   y.f++;
}
```

```
while (x.f>0){
   x.f--;
   y.f--;
}
```

As seen above, the loop on the left cannot be unconditionally proven to be terminating (actually it is not) because condition (1) does not hold. The analysis in COSTA considers all possible aliasing preconditions (in this case, there are only two possibilities $x = y$ and $x \neq y$) and transforms the fields into local variables according to the aliasing in the preconditions. It is thus able to prove termination and infer an upper bound for the second precondition. Aliasing preconditions are added in the method contract in a *//@requires* statement. In the example on the right, COSTA is able to prove termination and to infer an upper bound for both aliasing preconditions $x = y$ and $x \neq y$. However, COSTA is not able to prove it unconditionally. Straightforward post-processing is performed to transform the two cases into one unconditional termination annotation.

Example 22 The following annotated code is generated for the program on the left above. It contains the aliasing preconditions discussed above:

```
1 /*@ public behavior
2   @ requires x ≠ y;
3   @ ensures true;
4   @ signals_only Exception;
5   @ signals (Exception) true;
6   @ */
7 void m ()
8   //@ ghost int g0 = x.f; int g1 = y.f
9   //@ loop_invariant (g0 = x.f ∧ g1 = y.f)∨(g0 ≥ x.f+1 ∧ y.f ≥ g1 + 1);
10  //@ decreasing x.f ≥ 0 ? x.f : 0);
```

```
11  while (x.f>0){
12    x.f--;
13    y.f++;
14    //@ assert x.f = g0−1 ∧ y.f = g1+1
15    //@ set g0 = x.f ∧ g1 = y.f
16  }
17 }
```

COSTA managed to infer that the condition $x \neq y$ is to guarantee the existence of a ranking function (L10). This condition is added to precondition of the the method contract *//@ requires* $x \neq y$ (L2). To be able to state this precondition we used two ghost variables g_0 and g_1 (L8) for representing two different heap locations. Both variables are also used in the *//@loop_invariant* (L9) and in the *//@assert* (L14). ■

If wanted to discover several aliasing preconditions this would imply to run COSTA for each possible aliasing configuration. We have not found realistic examples that lead to a large number of configurations (referring to the experiments in [4]). We believe this is because experienced programmers avoid situations where aliasing of data could compromise termination and resource boundedness, that is, situations where the data involved in the termination proof can alias with other data modified within the loop.

4.2 Verification of Field-Sensitive Upper Bounds

In Section 3.2.1 we described the heap model used by KeY. Here, we focus on the extension of the heap model to deal with fields and in particular with aliasing. This part is needed to verify the field-sensitive upper bounds computed by static analysis.

Dereferencing the null pointer. The assignment rule in Section 3.2.1 was simplified in the sense that it did not consider the case where the object whose field was assigned a new value is `null`. The actual rule looks close to

$$\text{assign}\ \frac{\begin{array}{c}\Gamma \Longrightarrow \langle \mathsf{throw\ new\ NullPointerException();\ rest}\rangle\phi, \Delta \\ \Gamma \Longrightarrow \{\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, \mathtt{v})\}\langle \mathsf{rest}\rangle\phi, \Delta\end{array}}{\Gamma \Longrightarrow \langle \mathsf{o.f = v;\ rest}\rangle\phi, \Delta}$$

The first branch treats the case where the object is `null` and, consequently, a `NullPointerException` (NPE) is thrown. The second case assumes that the object reference was not the null reference and the object's field is assigned its new value. KeY provides the option to use an alternative set of assignment rules for objects for which one has to show in the first branch that the `NullPointerException` is not thrown. This second alternative is stronger than the first one, where we throw an NPE. It prevents a proof from being closed in the presence of implicitly thrown exceptions even if the postcondition to prove would be also valid otherwise. The general insight behind the alternative approach is that implicitly thrown exceptions are rarely intended by programmers and indicate programming errors.

In this article we use only the version where NPEs are actually thrown. The reason is that for the purpose of resource analysis we are not so much interested

in general programming errors. Instead, we want to have a guarantee about the worst-case execution behavior even when an NPE should be thrown, for instance when accessing a field.

Aliasing. For performance reasons in KeY aliasing analysis is postponed until the aliasing information is actually needed. For example, symbolic execution of `o.f = i; u.f = j;` does not cause a split because of aliasing.[3] The heap representation we end up with after symbolic execution is

$$store(store(\texttt{heap}, \texttt{o}, \texttt{f}, \texttt{i}), \texttt{u}, \texttt{f}, \texttt{j})$$

KeY has update simplification rules that can exploit for instance that one can show $o = u$. These rules would rewrite an update as the one above to $store(\texttt{heap}, \texttt{u}, \texttt{f}, \texttt{j})$, avoiding the need for aliasing checks later on.

The moment when KeY needs to handle aliasing comes not during symbolic execution, but when the value of a field is accessed in a logical term, e.g., to prove a sequent such as

$$\texttt{o} \neq \texttt{null}, \texttt{u} \neq \texttt{null} \Longrightarrow \{store(store(\texttt{heap}, \texttt{o}, \texttt{f}, \texttt{i}), \texttt{u}, \texttt{f}, \texttt{j})\}(\texttt{o.f} = \texttt{i}) \ .$$

Applying the update on the formula results in the sequent

$$\texttt{o} \neq \texttt{null}, \texttt{u} \neq \texttt{null} \Longrightarrow \mathit{if}\ (\texttt{o} = \texttt{u})\ \mathit{then}\ (\texttt{j})\ \mathit{else}\ (\mathit{if}\ (\texttt{o} = \texttt{o})\ \mathit{then}\ (\texttt{i})\ \mathit{else}\ (\texttt{o.f}) = \texttt{i}) \ .$$

The update application results in nested conditional terms whose guards represent aliasing checks. By delaying the aliasing decision until it is absolutely necessary and by aggressive application of simplification rules KeY can deal with aliasing rather efficiently and is able to remove many unnecessary splits. For the context of this article, which deals with reference data structures, it is frequently necessary to show that two different variables are not aliases. A fact that is often given directly by COSTA in terms of inequations or can be deduced by exploiting knowledge about acyclicity or disjointness of data structures.

Example 23 We look at one iteration of the loop in the previous example:

```
x.f--;
y.f--;
//@ assert x.f = g0−1 ∧ y.f = g1+1
```

During symbolic execution the postdecrement statements are rewritten into a sequence of assignments. Eventually, the assign rule shown above is executed twice. Each time the proof is split into two branches according to whether `x` and `y` are `null` or not.
Finally, the `assert` statement is on the branch on which both variables `x` and `y` are not null. The evaluation of the boolean assert expression is based on the evaluation of the expressions `x.f` and `y.f` in the reached heap state. The actual value(s) of both expressions differs depending on whether `x` and `y` are aliases or not. For instance, if `x` and `y` are aliases, then the assertion does not hold. Hence, the program calculus has to consider both cases and to split the proof accordingly.

[3] To be more precise, there is some splitting, but only to ensure that `o` and `u` are not `null`

We have the possibility to make use of the aliasing precondition inferred by COSTA to rule out the aliasing and maintain the precision to prove that the assertion is satisfied. Because `x` and `y` are not changed, we can rule out aliasing for *any* loop iteration. Hence, we can close branch (*ii*) of rule loopInv (see Section 2.3.3) and so verify that the assertion and loop invariant hold for any possible loop iteration. ■

5 Experimental Evaluation

To automate the integration of COSTA and KeY several non-trivial extensions have been made to both systems. The first two extensions are needed because COSTA works on Java bytecode, whereas KeY verifies Java source code:

1. Output the proof obligations using the original variable names (at the bytecode level, operand stack variables typically are used).
2. Place the annotations in the Java source code at the precise program points where they must be verified (entry points of loops).
3. To find a suitable JML format for representing proof obligations on UBs required several attempts (this concerns defining ghost variables, introducing `assert` constructs, etc.);
4. Implement the JML `assert` construct in KeY which was not supported hitherto.
5. Generate the heap proof obligations, including `\depth`, `\acyclic` and `\disjoint`.
6. For the heap-sensitive extension, identify the reference used to access those heap which have been transformed into local variables.
7. For programs whose termination depends on an aliasing condition, generate the transformed program and its corresponding annotations including the aliasing conditions inferred by COSTA.

The integrated tool can be used via the Eclipse plugins for both the extended COSTA and KeY systems which are available from `http://costa.ls.fi.upm.es/costa-key/`.

Table 1 shows a representative set of experiments which covers all features described in the article. The source code for all examples is available from the above website. The table is divided into three parts:

Integers aims at evaluating the behaviour of our technique for programs whose resource consumption depends on local Integer data. We evaluated several sorting algorithms, including bubble sort (`bubsort`), insertion sort (`inssort`), selection sort (`selsort`), and a program to sort the contents of a matrix (`msort`). We also evaluated a method to generate a Pascal Triangle (`pastri`) and a score board method (`sboard`) where two nested loops are used to initialize a matrix.

Heap Data Structures includes a set of programs that perform common operations on sorted lists as well as a method for searching an element in a binary tree. The name of the experiment describes the operation performed on the list.

Heap-Sensitive evaluates the behaviour of COSTA and KeY for handling programs whose UB depends on data stored in the heap. The first four experiments unconditionally terminate, `factorial` returns the factorial of a value stored in an object field, `fieldcomp` contains a loop that accesses two different fields to evaluate its condition, `nestloops` has two nested loops which access the same

Bench	Certificate Generation							Checking	
	COSTA			KeY			Total		
	$\mathbf{T}_{heap}$	$\mathbf{T}_{UB}$	$\mathbf{T}_{jml}$	$\mathbf{T}_{ver}$	**Nod**	**Br**	$\mathbf{T}_{gen}$	$\mathbf{T}_{check}$	%
Integers									
`bubsort`	14	278	12	12100	4364	75	12390	3300	26.63
`inssort`	14	175	9	14800	4708	71	14984	2600	17.35
`msort`	16	238	12	22900	6747	147	23150	3900	16.85
`selsort`	11	230	10	11800	4286	55	12040	3400	28.24
`pastri`	36	468	20	70100	13884	222	70588	10800	15.30
`sboard`	11	152	8	7700	2770	55	7860	2700	34.35
Heap Data Structures									
`traverse`	11	47	4	1000	1208	52	1051	1000	95.15
`create`	36	134	8	2500	1638	52	2642	1100	41.64
`insert`	239	406	18	40000	17485	586	40424	10100	24.99
`indexOf`	23	150	5	5400	2439	67	5555	2800	50.41
`reverse`	50	110	9	25600	14270	678	25719	3600	14.00
`array2List`	43	146	8	2900	1442	37	3054	1400	45.84
`copy`	49	111	9	19600	14175	679	19720	2800	14.20
`searchtree`	111	194	7	2700	2389	97	2901	1500	51.71
Heap-Sensitive									
`factorial`	4	65	3	1000	643	13	1068	1000	93.63
`fieldcomp`	8	99	8	2400	1027	13	2507	1400	55.84
`nestloops`	9	152	8	4600	2026	27	4760	1900	39.92
`arrayelem`	5	71	4	3200	1443	41	3275	1500	45.80
`basiccond`	8	174	9	1300	823	25	1483	1100	74.17
`noteqcond`	9	209	10	2700	1560	38	2919	1400	47.96
`multicond`	8	393	11	2900	1567	37	3304	1600	48.43
`nestedlcond`	16	921	23	10100	4084	74	11044	2700	24.45
`multiloop`	16	425	21	5400	2517	64	5846	2000	34.21

Table 1 Statistics for the Generation and Checking of Resource Guarantees (times in ms)

field, and `arrayelem` whose termination depends on one element of an array declared as an object field. Furthermore, this set includes five programs to evaluate our technique for programs whose UB can only be obtained under aliasing preconditions inferred by COSTA: `basiccond` depends on the condition `x=y` where `x` and `y` are references, `noteqcond` depends on the condition `x`$\neq$`y`, `multicond` termination requires conditions over three variables, `nestedlcond` contains two nested loops with different conditions which must be combined, and `multiploop` also contains two loops but in this case they are not nested.

Columns $\mathbf{T}_{heap}$, $\mathbf{T}_{UB}$, $\mathbf{T}_{jml}$, show, respectively, the times (in ms) taken by COSTA to apply the heap analysis (i.e. path-length, cyclicity and sharing), the overall resource analysis (including the heap analysis) and to generate the JML annotations. Column $\mathbf{T}_{ver}$ shows the time (in ms) taken by KeY to verify the JML annotations generated by COSTA. We show in addition the number of nodes (**Nod**) and branches (**Br**) of the generated proof to provide some insight on the proof complexity. Column $\mathbf{T}_{gen}$ shows the time taken to generate the proof ($\mathbf{T}_{UB}$ +

$\mathbf{T}_{jml} + \mathbf{T}_{ver}$). Proofs obtained by KeY can be saved and used later as a certificate. Column $\mathbf{T}_{check}$ shows the time (in ms) taken by KeY to check the validity of the correctness proof, and column % shows the ratio between the generation of the proof and the time needed to validate it ($\mathbf{T}_{check}/\mathbf{T}_{gen}$). All times are measured in ms and were obtained using an Intel Core i5 at 1.8GHz with 4Gb of RAM running a MacOS 10.8 (Mountain Lion).

Our experiments demonstrate 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, the inference process of COSTA is faster than verification with KeY, due to the inherent complexity of formal verification compared to static analysis. On the other hand, checking an existing proof with KeY is faster than the time needed to produce it. This reduction in the time taken to check and the validity of the proof is more significant in those experiments with a greater number of nodes. This behaviour can be observed in the experiments `msort`, `insert`, `reverse` or `copy`, all of them with a proof of more than 5000 nodes and with a checking time under 25% of the time taken by the verification phase.

The certificates generated are quite large, but we could apply existing techniques to reduce the size of certificates. For instance, there is work that studies certificate size reduction for proofs generated by abstract interpretation-based analyzers [7]. The main idea is to include in the certificate that has to be sent only those fragments of the proofs that require iterating for finding a fixed point. In these cases, the fixed point of the analysis is sent in the certificate and the checker has to perform a single iteration on the fixed point to prove its validity. The other fragments of the proof can be found by a non-iterative checking algorithm and thus do not have to be transmitted. This idea could be also applied to the proofs obtained to certify resource boundedness. But to do this, one needs to create an efficient one-pass checker. This is outside the scope of this article.

Another interesting result of our work is that, during the experimental evaluation, KeY was able to spot a bug in COSTA, as it failed to prove correct one invariant which was incorrect. In addition, KeY could provide a concrete counterexample that helped understand, locate and fix the bug, which was related to a recently added feature of COSTA.

6 Related Work

There exist several other cost analyzers which automatically infer resource guarantees for different programming languages [23, 24]. However, none of them formally proves the correctness of the upper bounds they infer. An exception is [20], which verifies and certifies resource consumption (for a small programming language and not for heap properties). For the particular case of memory resources, [21] 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. Interestingly, this approach, though weaker in principle than verification of the analyzer, has advantages in the context of mobile code because generated proofs

can act as resource certificates. Following proof-carrying-code [29] principles, code originating from an untrusted *producer* can be bundled together with the proof generated by KeY for its declared resource consumption. This way, the code *consumer* can check locally and automatically using KeY whether the claimed resource guarantees are verified.

Many software verification tools including KeY [15], Why [22], VeriFast [33], or Dafny [25] 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. This confirms the findings of the SLAM project [13] that existing verification technology can be highly automatic for realistic programs and a restricted class of properties.

Recent work studies the formal verification of loop bounds in the context of WCET analysis [17], which can be considered as a particular type of resource. This approach is based on the formal verification of the compiler [26] and relies on program slicing and bound calculation to annotate the source code with loop bounds information. The approach is limited to the verification of loop bounds, while our framework can handle not only loop bounds but it can also verify size relations between different parts of the source code and the more general property of the resource consumption of the program.

7 Conclusions and Future Work

This paper describes a framework for the combination of static analysis and formal verification that allows to formally verify the results of static analysis tools. The combination is based on the notion of method contracts which enable (1) a method-by-method translation of static analysis results into verification proof obligations and (2) render verification modular and hence scalable as well as . We proved the viability of our approach by instantiating it to the case of automated resource analysis.

Our tool can handle not only simple programs whose resource consumption depends on integer data, but also those that depend on the size of heap-allocated data structures. The distribution of work among the two systems is as follows: the resource analyzer generates ranking functions, invariants, as well as size relations, and outputs them as extended JML annotations of the analyzed program; the formal verification tool then verifies the resulting proof obligations in its program logic and produces proof certificates that can be saved and reloaded. To the best of our knowledge, this is the first time that generic resource bounds for a realistic object-oriented programming language are automatically certified. Our work paves the way for a proof-carrying code approach to resource analysis.

It is important to note that the framework presented in this paper is developed for the classical approach to cost analysis [35]. The underlying analyses are classified into two categories: (1) those that compute intermediate information such as ranking functions and invariants; and (2) the algorithm that uses the intermediate results to compute the upper bounds. We argued that verifying the implementation of point (2) is rather simple, and can be done even manually. The challenge is

to guarantee the correctness of the intermediate results, which we do by verifying them using KeY instead of verifying the corresponding implementations.

Following a similar methodology one can develop verification tools for bounds generated by other cost analysis frameworks as follows: (1) one needs first to distinguish simple components whose implementation can be verified easily, from complex ones that require a considerable effort to be verified; and (2) instead of verifying the implementations of the complex components we can use a theorem prover, as we have done using KeY. For example, for SPEED [23] one would need to verify the correctness of the generated invariants, while using other techniques to verify the the implementation of the other parts, e.g., the implementation of the Counter-Optimal Proof-Structure procedure. As another example we consider the extension of COSTA, described in [11], to handle cases where the cost can be precisely described using a sum such as $\Sigma_{i=0}^{n} i$. In this case the algorithm for computing the UB is based on using the ranking functions and invariants to first generate recurrence relations, and then solve them using computer algebra systems. This algorithm is quite simple, and can be verified even manually; however, the complex components are those that generate ranking functions and loop invariants which we already handle in this paper.

It is clear that COSTA might fail to infer bounds for some programs, since the problem of inferring cost is undecidable in general (see [6] for more details). However, there are classes of programs for which COSTA is able to infer upper bounds that cannot be verified in our framework. These include recursive programs, where ranking functions do not represent bounds on the number of loop iterations, but on the depth of recursion which (at the moment) cannot be handled directly in KeY. A possible solution is to verify an invariant that describes how the method's parameters change when called recursively, and then use this invariant that quantifies the change to prove that the ranking function indeed bounds the recursion depth. This is left for future work.

Also as future work, we plan to consider the verification of resource bounds for *concurrent* programs. There is recent work on the inference of resource bounds for actor-based concurrent programs [12], a model of concurrency that is simpler and more modular than Java threads. Even in this simpler setting, the inference of upper bounds requires a pre-analysis in which one infers the concurrent interleavings (also known as may-happen-in-parallel analysis) that can occur in the execution. This information is essential to obtain sound upper bounds, since task interleavings can lead to larger bounds and thus they must be considered to produce sound approximations of the worst-case cost. To verify these bounds, one therefore must also verify the results of the analysis that infers the concurrent interleavings.

Acknowledgements We gratefully thank the anonymous referees for many useful comments and suggestions that greatly helped to improve this article. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO project TIN2012-38137, and by the CM project S2013/ICE-3006.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *Proc. of FMOODS'08*, volume 5051 of *LNCS*, pages 2–18.

Springer, 2008.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Field-Sensitive Value Analysis by Field-Insensitive Analysis. In *Proc. of FM'09*, volume 5850 of *LNCS*, pages 370–386. Springer, 2009.
3. 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.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and G. Román-Díez. Conditional Termination of Loops over Heap-allocated Data. *Science of Computer Programming*, 92:2 – 24, 2014.
5. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*. Springer, 2007.
6. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159, 2012.
7. E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo. Certificate Size Reduction in Abstraction-Carrying Code. *Theory and Practice of Logic Programming*, 12(3):283–318, May 2012.
8. E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. Verified Resource Guarantees using COSTA and KeY. In *Procs. of PEPM'11*, pages 73–76, 2011.
9. E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-Díez. Verified Resource Guarantees for Heap Manipulating Programs. In *Procs. of FASE'12*, volume 7212 of *LNCS*, pages 130–145. Springer, March 2012.
10. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proc. of ISMM'10*, pages 121–130. ACM Press, 2010.
11. E. Albert, S. Genaim, and A. N. Masud. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Transactions on Computational Logic*, 14(3):22:1–22:35, 2013.
12. E. Albert, P.Arenas, J. Correas, M. Gómez-Zamalloa, S. Genaim, G. Puebla, and G. Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects, Technical Report. http://costa.ls.fi.upm.es/papers/costa/AlbertACGGPRtr.pdf. 2014.
13. T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The Static Driver Verifier research platform. In *Proc. of CAV'10*, volume 6174 of *LNCS*, pages 119–122. Springer, 2010.
14. M. Barnett, B. Chang, R. DeLine, B. Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Procs. of FMCO'06*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.
15. B. Beckert, R. Hähnle, and P. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.
16. D. Beyer and M.Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.
17. S. Blazy, A. Maroneze, and D. Pichardie. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Procs. of VSTTE'13*, volume 8164 of *LNCS*, pages 281–303. Springer-Verlag, 2013.
18. M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 413–429. Springer Berlin Heidelberg, 2013.
19. R. Bubel, A. Roth, and P. Rümmer. Ensuring the correctness of lightweight tactics for JavaCard dynamic logic. *Electron. Notes Theor. Comput. Sci.*, 199:107–128, 2008.
20. K. Crary and S. Weirich. Resource Bound Certification. In *Procs. of POPL'00*, pages 184–198. ACM, 2000.
21. De Dios and R. Peña. Certification of Safe Polynomial Memory Bounds. In *Proc. of FM'11*, LNCS, pages 184–199. Springer, June 2011.
22. J.C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.
23. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.
24. J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *Procs. of ESOP'10*, volume 6012 of *LNCS*, pages 287–306. Springer, 2010.

25. K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proc. of LPAR'10*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
26. X. Leroy. Formal verification of a realistic compiler. *Communications ACM*, 52(7):107–115, 2009.
27. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
28. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 5–20. Springer-Verlag, 2005.
29. G. Necula. Proof-Carrying Code. *Procs. of POPL'97*, pages 106–119. ACM Press, 1997.
30. A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Proc. of TACAS'98*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.
31. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, LNCS, pages 239–251. Springer, 2004.
32. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Proc. of SAS'05*, volume 3672 of *LNCS*, pages 320–335. Springer, 2005.
33. J. Smans, B. Jacobs, F. Piessens, and Schulte W. An Automatic Verifier for Java-like programs based on dynamic frames. In *Proc. of FASE'08*, volume 4961 of *LNCS*, pages 261–275. Springer, 2008.
34. F. Spoto, P.M. Hill, and E. Payet. Path-Length Analysis of Object-Oriented Programs. In *Proc. of EAAI'06*, 2006. Available at http://profs.sci.univr.it/ spoto/papers.html.
35. B. Wegbreit. Mechanical Program Analysis. *Communications ACM*, 18(9):528–539, 1975.
36. B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.

A JML Annotations Overview

COSTA	Sec.	JML
Size Relations φ Initial variables $\overline{x}_0$ Variables $\overline{x}$	2.2	*//@ghost int* $x_0^i{=}x^i; \ldots;$ *int* $x_0^n{=}x^n;$ *//@set* $x_0^i{=}x^i; \ldots; x_0^n{=}x^n;$ *//@assert* φ
Ranking Function: $\#iter = \mathsf{nat}(f)$	2.2	*//@decreasing f ? f : 0;*
Loop Invariant ψ Initial variables $\overline{x}_0$ Variables $\overline{x}$	2.2	*//@ghost int* $x_0^i{=}x^i; \ldots;$ *int* $x_0^n{=}x^n;$ *//@loop_invariant* ψ
I/O Size Relations φ_{io}	2.2	*//@ public behavior* *//@ requires true;* *//@ ensures* φ_{io}*;* *//@ signals_only Exception;* *//@ signals (Exception) true;*
Path Length of variable x	3.1.1	`\depth(x)`
x is acyclic	3.1.2	`\acyclic(x)`
x might reach y in zero or more steps	3.1.2	`\reach(x,y)`
x might reach y in one or more steps	3.1.2	`\reachPlus(x,y)`
x and y do not share	3.1.3	`\disjoint(x,y)`