Verified Resource Guarantees using COSTA and KeY

Elvira Albert
Complutense University of Madrid
elvira@sip.ucm.es

Richard Bubel
Chalmers University of Technology
bubel@chalmers.se

Samir Genaim
Complutense University of Madrid
samir.genaim@fdi.ucm.es

Reiner Hähnle
Chalmers University of Technology
reiner@chalmers.se

Germán Puebla
Technical University of Madrid
german@fi.upm.es

Guillermo Román-Díez
Technical University of Madrid
groman@fi.upm.es

Abstract

Resource guarantees allow being certain that programs will run within the indicated amount of resources, which may refer to memory consumption, number of instructions executed, etc. This information can be very useful, especially in real-time and safety-critical applications. Nowadays, a number of automatic tools exist, often based on type systems or static analysis, which produce such resource guarantees. In spite of being based on theoretically sound techniques, the implemented tools may contain bugs which render the resource guarantees thus obtained not completely trustworthy. Performing full-blown verification of such tools is a daunting task, since they are large and complex. In this work we investigate an alternative approach whereby, instead of the *tools*, we formally verify the *results* of the tools. We have implemented this idea using COSTA, a state-of-the-art static analysis system, for producing resource guarantees and KeY, a state-of-the-art verification tool, for formally verifying the correctness of such resource guarantees. Our preliminary results show that the proposed tool cooperation can be used for automatically producing verified resource guarantees.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]; D3.0 [*Programming Languages*]

General Terms Languages, Theory, Verification, Reliability

Keywords Static Analysis, Resource Guarantees, Java

1. Introduction

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of systems. Verifying the correctness of modern static analyzers is rather challenging, among other things, because of the sophisticated algorithms used in them, their evolution over time, and, possibly, proprietary considerations. A simpler alternative is to construct a validating tool [7] which, after every run of the analyzer, formally confirms that the results are correct and, optionally, generates correctness proofs. Such proofs could then be translated to *resource certificates* [5, 6].

PEPM'11, January 24–25, 2011, Austin, Texas, USA.

In this work, we are interested in *resource guarantees* obtained by static analysis. An essential aspect of programs is that resources be used effectively. This is especially true in the current programming trends, which provide us with mechanisms for code reuse by means of components and services: not only functionality, but also resource consumption (or *cost*) must be taken into consideration.

COSTA is a state-of-the-art COSt and Termination Analyzer for Java bytecode (and hence Java). It 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 [3], memory consumption, or calls to certain method) 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 [2]. Intuitively, this requires (1) bounding the number of iterations of each loop and (2) finding the worst-case cost among all iterations. *Ranking functions* [8] 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 in 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 main components of the UB generation were proven correct at a theoretical level. However, there is no guarantee that correctness is preserved in the actual implementation which is rather involved.

KeY [4] is a state-of-the-art 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. The main contribution of this work is to show that, using KeY, it is possible to formally and automatically verify the correctness of the UBs obtained by COSTA.

2. Inference of Upper Bounds in COSTA

In this section, we briefly describe the techniques used in COSTA for automatically inferring UBs, and we identify the proof obligations that need to be verified using KeY.

2.1 Main Components of an Upper Bound

Consider the following (JML annotated) program that implements the insert sort algorithm.

```
1 void insert_sort (int A[]) {
2   int i, j, v;
3   //@ ghost int i0=i; int j0=j; int a0=a;
4   i=A.length-2;
5   //@ assert (i=i0-2 ∧ j=j0 ∧ a=a0)
6   //@ ghost int i1=i; int j1=j; int a1=a;
7   //@ loop_invariant i≤i1
8   //@ decreases i>0 ? i : 0
9   while ( i>=0 ) {
10    //@ ghost int i2=i; int j2=j; int a2=a;
11    j=i+1;
12    v=A[i];
13    //@ assert j=i2+1 ∧ i2 ≥ 0
14    //@ ghost int i3=i; int j3=j; int a3=a;
15    //@ loop_invariant j≤a3
16    //@ decreases a - j>0 ? a - j : 0
17    while ( j<A.length && A[j]<v) {
18      A[j-1]=A[j];
19      j++; }
20    A[j-1]=v;
21    i--; } }
```

COSTA receives a non-annotated version of the above program and, for the cost model that counts the number of executed bytecode instructions, produces the (asymptotic) UB insert_sort(a)=a^2, where a refers to A.length. The underlying analysis used in COSTA infers UBs for each iterative and recursive constructs (loops) and then composes the results in order to obtain an UB for the method of interest. Intuitively, in order to infer an UB for a single loop, it first infers an UB A on the cost of a single execution of its body, an UB I on the number of iterations that it can make, and then $A * I$ is an UB for the loop. In order to infer A and I COSTA relies on several program analysis components that provide essential information:

Ranking functions. For each loop, COSTA infers a linear function from the loop variables to $\mathbb{N}$ which is decreasing at each iteration. For example, for the loop at line 17, it infers function $f(a,j) = \mathsf{nat}(a-j)$ where $\mathsf{nat}(\ell) = \max(0,\ell)$. This function can be safely used to bound the number of iterations. In the example, if a_3 and j_3 are the initial values of a and j, then it is guaranteed that $f(a_3, j_3)$ is an UB on the number of iterations of the loop.

Loop invariants. For each loop in the program, COSTA infers an invariant that involves the loop's variables and their initial values (i.e., their values before entering the loop). Let us denote by i_1 the initial value of i when entering the loop at line 9. COSTA infers the invariant $i \leq i_1$, which states that i is always smaller than or equal to its initial value when the program reaches the loop condition. This information, together with the size relations below, is needed to compute the worst-case cost of executing one loop iteration.

Size relations. Given a fragment of code or a scope (details below), COSTA infers relations between the values of the program variables at a certain program point of interest within the scope and their initial values when entering the scope. For example, at program point 13, it infers that $j = i_2 + 1$, where i_2 is the value of i when entering the scope that contains line 13 (i.e., the scope here is the loop body). In this case the relation is a simple consequence of the instruction at line 11. In general, however, it may not be trivial to infer such relations nor to prove that they are correct.

Upper Bounds. Once the above information has been inferred, it is straightforward to compute an UB for the method. Let us show this process on the running example:

Inner loop. The process starts from the innermost loops. Thus, we start with the loop at line 17. Assuming that executing the condition costs (at most) c_1 instructions, and that the cost of each iteration (i.e., the loop body) is c_2 instructions, then it is clear that $\mathsf{nat}(a_3 - j_3) * (c_1 + c_2) + c_1$ is an UB on the cost of this loop (because c_1 and c_2 are constant).

Outer loop. Next, we move to the outer loop at line 9. Let us assume that the cost of the comparison is c_3 instructions, the code at lines 11–12 are c_4 instructions, and the code at lines 20–21 are c_5 instructions. Then, the cost of each iteration of this loop is $c_3 + c_4 + \mathsf{nat}(a_3 - j_3) * (c_1 + c_2) + c_1 + c_5$, where the highlighted subexpression corresponds to the cost of the inner loop computed above. Note that in this case, each iteration might have a different cost, since $a_3 - j_3$ is not the same for all iterations. Simply multiplying the number of iterations $\mathsf{nat}(i_1)$ by such a cost is unsound. The solution is to find an expression U in terms of the initial values of a_1, i_1, j_1 which does not change during the loop such that $U \geq a_3 - j_3$ in all iterations. Then, $\mathsf{nat}(i_1) * [c_3 + c_4 + \mathsf{nat}(U) * (c_1 + c_2) + c_1 + c_5] + c_3$ is an UB for the loop. In order to find such a U, COSTA uses the loop invariant (line 7) and the size relations (line 13) as follows: it solves the parametric integer programming (PIP) problem of maximizing the objective function $a_3 - j_3$ w.r.t. the loop invariant and the size relations where i_1, a_1, j_1 are the parameters. This produces an expression in terms of i_1, a_1, j_1 which is greater than or equal to $a_3 - j_3$ in all iterations of the loop. In our example, it is $U = a_1 - 1$.

Method. We finally can compute the cost of the insert_sort method. Assume that the cost of line 4 is c_6, then the cost of the method is $c_6 + \mathsf{nat}(i_1) * [c_3 + c_4 + \mathsf{nat}(a_1 - 1) * (c_1 + c_2) + c_1 + c_5] + c_3$. We need to express this UB in terms of the input parameter a. For this, COSTA maximizes (using PIP) i_1 and $a_1 - 1$ w.r.t. the size relation at line 5 and, respectively, obtains $a-2$ and $a-1$. Therefore, $c_6 + \mathsf{nat}(a-2) * [c_3 + c_4 + \mathsf{nat}(a-1) * (c_1 + c_2) + c_1 + c_5] + c_3$ is the UB for insert_sort.

2.2 COSTA Claims as JML Annotations

To justify that the UBs obtained by COSTA are correct, we need to provide formal correctness proofs for all the claims above. This includes the ranking functions, invariants, size relations, the cost model that provides all c_i, and the underlying PIP solver.

Correctness of the cost model is trivial as it is a simple mapping from each instruction to a number. Correctness of the underlying PIP solver is also straightforward if we use the maximization procedure defined in [2], which is based only on the Gaussian elimination algorithm. Therefore, we concentrate on verifying the correctness of the ranking functions, size relations and invariants. They are inferred by large software components whose correctness has not been verified. We now briefly describe the translation of the different pieces of information generated by COSTA to JML annotations on the Java program, which will allow their verification in KeY.

Ranking functions. For a given loop, when COSTA infers a ranking function of the form $\mathsf{nat}(\ell)$, we translate it to the JML annotation "//@ decreasing $\ell > 0$? ℓ : 0", since $\mathsf{nat}(\ell)$ can be defined as an if-then-else. COSTA might provide also ranking functions of the form $\log(\mathsf{nat}(\ell) + 1)$, which are handled similarly.

Invariants. COSTA infers an invariant φ for each loop. This invariant involves the loop variables $\bar{v}$ and auxiliary variables $\bar{w}$ such that each w_i represents the initial value of v_i. The JML annotation

for this invariant consists of one line defining all $\bar{w}$ as ghost variables ("//@ ghost int $w_1 = v_1$;...; int $w_n = v_n$") and one line for declaring the loop invariant ("//@ loop_invariant φ").

Size relations. Size relations are linear constraints between the values of a set of variables of interest between two program points. As we have seen, this allows composing the cost of the different program fragments. For each loop (or method call), COSTA infers the relation φ between the values before the loop entry (or the call) and the entry of its parent scope. Suppose that the loop (or the call) is at line L_l, its parent scope starts at line L_p, and that $\bar{v}$ are the variables of interest at L_l and $\bar{w}$ represent their values at L_p. Then we add the JML annotation "//@ ghost int $w_1 = v_1$;... ; int $w_n = v_n$;" immediately after line L_p to capture the values of $\bar{v}$ at line L_p, and the JML annotation "//@ assert φ" immediately before line L_l to state that the relation φ must hold at the program point. Additional size relations inferred by COSTA are input-output size relations. These are linear constraints that relate the return value of a given method to its input values. For example, suppose that we replace "i −−" in line 21 of the insert_sort program by "i =decrement(i)" where decrement is defined by "**int** decrement(**int** x) {**return** x−1;}". Then COSTA infers the relation "$\varphi \equiv$\result= x−1" which is used to bound the number of iterations of that loop. In order to verify this relation in KeY we add the JML annotation "//@ ensures φ" to the contract of decrement.

3. Verification of Upper Bounds using KeY

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

3.1 Verification by Symbolic Execution

The program logic used by KeY is *JavaCard Dynamic Logic* (JavaDL) [4], a first-order dynamic logic with arithmetic. Programs are first-class citizens similar to Hoare logics 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

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

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

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

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 in order 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 wrt ϕ.

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 insert_sort this formula (using the default contract) is:

$$\forall o; \forall a0; \{\mathsf{a} := a0 \parallel \mathsf{self} := o\}(\neg(\mathsf{a} \doteq \mathsf{null}) \wedge \neg(\mathsf{self} \doteq \mathsf{null}) \rightarrow \langle\ \mathsf{try\ \{\ self.insert\_sort(a)@NestedLoops;\ \}\ catch(Exception\ e)\{\ exc=e;\ \}}\rangle(\mathsf{exc} \doteq \mathsf{null} \vee \mathsf{instance}_{\mathsf{Exception}}(\mathsf{exc}))$$

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

3.3 Verification of Proof-Obligations

The proof obligation formula must be proven valid by executing the method insert_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. In the insert_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 Sect. 2.2). Input-output size relations encoded in terms of method contracts are proven correct as outlined in Sect. 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:

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

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 monotonic while remaining non-negative. Premise (iii) continues symbolical execution upon loop exit. The integer-typed variant term ensures loop termination as it has a lower bound (0) and is decreased by each loop iteration. Using COSTA's derived ranking function as variant term obviously verifies that the ranking function is correct. The update $\mathcal{U}_A$ assigns to all locations whose values are potentially changed by the loop a fixed, but unknown value. This allows using the values of locations that are unchanged in the loop during symbolic execution of the body.

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 are. 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. 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 occuring normalisation steps.

4. Implementation and Experiments

The implementation of our approach has required the following non-trivial extensions to COSTA and KeY (note that COSTA works on Java bytecode, and KeY on Java source): (1) output the proof obligations using the original variable names (at the bytecode level, operand stack variables are often used); (2) place the obligations in the Java source at the precise program points where they must be verified (entry points of loops); (3) finding a suitable JML format for representing proof obligations on UBs has required a considerable number of iterations (defining ghost variables, introducing assert constructs, etc.); (4) implement the JML assert construct in KeY which was not supported hitherto. To express assertions which have to hold before a method call but after parameter binding support for a second assertion construct invocAssert has been added.

Eclipse plugins for both the extended COSTA and KeY systems are available from `http://pepm2011.hats-project.eu`. Source code for the tools (under GPL) is planned in the near future.

Table 1 shows some preliminary experiments using a set of representative programs, available from the above website, which include sorting algorithms, namely bubble sort (*bubsort*), insert sort (*inssort*), and selection sort (*selsort*); a method to generate a Pascal Triangle (*pastri*); simple (*slm*) and nested loops (*nlf*). Columns $\mathbf{T}_{size}$, $\mathbf{T}_{inv}$, $\mathbf{T}_{rf}$, $\mathbf{T}_{ana}$ and $\mathbf{T}_{jml}$ show, respectively, the times taken by COSTA to obtain the size relations, loop invariants, ranking functions, the whole analysis (which includes the previous times) and generate the JML annotations. Column $\mathbf{T}_{ver}$ shows the time taken by KeY in order to verify the JML annotations generated by COSTA. As time measurements for Java are imprecise we state in addition the number of nodes and branches of the generated proof to provide some insight on the proof complexity. Column **Total** shows the time taken by the whole process. All times are measured in ms and were obtained using an Intel Core2 Duo P8700 at 2.53GHz with 4Gb of RAM running a Linux 2.6.32 (Ubuntu Desktop). A notable result of our experiments is that 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.

Bench	COSTA					KeY			**Total**
	$\mathbf{T}_{size}$	$\mathbf{T}_{inv}$	$\mathbf{T}_{rf}$	$\mathbf{T}_{ana}$	$\mathbf{T}_{jml}$	**Nodes**	**Branches**	$\mathbf{T}_{ver}$	
slm	22	20	26	112	4	3641	36	6700	6816
nlf	30	16	24	106	6	5665	37	2800	2912
bubsort	38	24	144	296	14	14890	230	57800	58110
inssort	30	12	46	142	6	9875	167	29300	29448
selsort	40	20	112	232	8	12564	209	40700	40940
pastri	66	38	138	394	14	29723	337	110100	110508

Table 1. Statistics about the Analysis and Verification Process

5. Conclusions and Future Work

We have demonstrated that automatic verification of the upper bounds inferred by COSTA using KeY is feasible. Instead of verifying the correctness of the underlying static analysis, we take the alternative approach of verifying the correctness of their results. Interestingly, this approach, though weaker in principle than verification of the analyzer, has advantages in the context of mobile code. Following proof-carrying-code [6] 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. As future work, we plan to extend our approach to support programs that manipulate data structures other than arrays.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-231620 *HATS* project, by TIN-2008-05624 *DOVES*, by UCM-BSCH-GR58/08-910502 (GPD-UCM) and S2009TIC-1465 *PROMETIDOS* project.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS'08*, volume 5051 of *LNCS*, pages 2–18. Springer, 2008.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 2010. To appear.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.

[4] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.

[5] K. Crary and S. Weirich. Resource Bound Certification. In *POPL'05*, pages 184–198. ACM Press, 2000.

[6] G. Necula. Proof-Carrying Code. In *POPL 1997*. ACM Press, 1997.

[7] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *TACAS'98*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

[8] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI'04*, LNCS. Springer, 2004.