CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

ResAna: A Resource Analysis Toolset for (Real-Time) Java

R. W. J. Kersten[1]*, B. E. van Gastel[2], O. Shkaravska[3], M. Montenegro[4] and M. C. J. D. van Eekelen[1,2]

[1] *Institute for Computing and Information Sciences, Radboud University Nijmegen*
[2] *School of Computer Science, Open University of the Netherlands*
[3] *Max Planck Institute for Psycholinguistics, Nijmegen*
[4] *Departamento de Sistemas Informáticos y Computación, Universidad Complutense Madrid*

SUMMARY

For real-time and embedded systems limiting the consumption of time and memory resources is often an important part of the requirements. Being able to predict bounds on the consumption of such resources during the development process of the code can be of great value. In this paper we focus mainly on memory related bounds.
Recent research results have advanced the state of the art of resource consumption analysis. In this paper we present a toolset that makes it possible to apply these research results in practice for (real-time) systems enabling JAVA developers to analyse symbolic loop bounds, symbolic bounds on heap size and both symbolic and numeric bounds on stack size. We describe which theoretical additions were needed in order to achieve this.
We give an overview of the capabilities of the RESANA toolset that is the result of this effort. The toolset can not only perform generally applicable analyses, but it also contains a part of the analysis which is dedicated to the developers' (real-time) virtual machine, such that the results apply directly to the actual development environment that is used in practice. Copyright © 2013 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Resource analysis, Polynomial interpolation, Ranking function, Loop bounds, Heap bounds, Stack bounds

1. INTRODUCTION

Both in industry and in academia there is an increasing interest in more detailed resource analysis bounds than orders of complexity. In correctness verification for industrial critical systems, the focus is often mainly on functional correctness: does the program deliver the right output with the right input. However, for such systems it is just as important to make sure that bounds for the consumption of time and space are not exceeded. Otherwise, a program may not react within the required time or it may run out of memory and come to a halt (making it vulnerable to a Denial Of Service attack).

Traditionally, the focus has been on performance analysis taking time as resource which is consumed. More recently, several researchers have produced significant results in heap and stack bound analysis. In this paper we focus on such *memory related resource analysis*. The symbolic loop bound analysis part however may be used both for memory and for time analysis.

*Correspondence to: Institute for Computing and Information Sciences, Digital Security, Mailbox number 47, Faculty of Science, University of Nijmegen, Postbus 9010, 6500GL Nijmegen, The Netherlands. E-mail: R.Kersten@cs.ru.nl

Contract/grant sponsor: ARTEMIS Embedded Computing Systems Initiative; contract/grant number: ARTEMIS-2008-1-100039 (CHARTER)

Many real-time and embedded systems critically depend on operating within a fixed amount of memory. Clearly, for such systems it can be important to know an upper bound on the consumed memory. For safety critical applications it can be essential. Programmers may be able to guess a bound and to prove it by hand. That activity is quite tedious and error-prone. A tool that in many cases is able to automatically infer bounds and proof them may be very helpful in the software development process. This paper presents such a tool.

For safety-critical applications often domain specific programming languages are used that have strong support for loop bounding or regular programming languages with strict coding conventions. In the recently finished EU Artemis CHARTER (**C**ritical and **H**igh **A**ssurance **R**equirements **T**ransformed through **E**ngineering **R**igour) project, REALTIME JAVA was considered as possible programming language for safety-critical systems. Reasons for studying REALTIME JAVA include more possibilities for code reuse, more available tools and more programmers that are highly experienced in the use of the language. The RESANA toolset, which is presented in this paper, is one of the results of the CHARTER project [1, 2, 3]. Together, the tools produced by the CHARTER project provide a first step towards the use of general programming languages for safety-critical systems. For full deployment in safety-critical context the CHARTER tool chain should be advanced further. For now, the RESANA toolset can already be used in everyday practice, e.g. for inferring and proving memory consumption properties of existing library functions and of non-critical applications for which memory bounds are relevant like applications for mobile devices. Another usage may be the development of prototype applications with verified resource consumption properties. These prototypes can then be transformed to the language that is in actual use for the safety-critical system. The techniques presented in this paper can in principle be used for other languages too. Of course, that would require both an adaptation of the front-end of the tool and of the annotation language that is used for expressing the properties.

Even if memory is abundantly available, applications can be hindered significantly when more memory is consumed than expected. Effectively the system may come to a halt due to excessive swapping. Some Denial-Of-Service attacks are based on this principle. A known upper bound of consumed memory may prevent attacks of that kind.

A variety of different memory analysis techniques have been developed independently not only on the language level but also on the byte code level [4]. Researchers use polynomial interpolation [5], reachability-bound analysis [6], amortization [7], polynomial quasi-interpretation [8] and new language features such as programmer-controlled destruction and copying of data structures [9]. Of course, such analyses are undecidable in general. In practice, however, an increasingly large set of problems can be handled.

This research builds upon earlier resource analysis work developed in the Dutch NWO AHA project [10]. In this paper, we focus on the JAVA language and on resource consumption properties related to heap and stack usage. Using the scoped memory which is offered by REALTIME JAVA one can enforce constant memory bounds and facilitate simple memory management. However, in order to deal with more complex bounds, a more thorough analysis is needed. While our research mainly focuses on REALTIME JAVA, the techniques and the tool described here are also applicable to regular JAVA programs. The loop bound analysis provided by the RESANA tool can be of further use both for deriving memory bounds and for deriving time bounds. This article is an extended version of an earlier paper, [3]. How this paper extends [3] is described in Section 6.

With the goals of making these results applicable in practice, our heap and stack resource analysis goes beyond orders of complexity. We aim at obtaining bounds that are expressions of relevant variables and parameters. If a resource is consumed quadratically with respect to the value of a parameter x, than a typical bound could be e.g. $2x^2 - 4x + 15$ thus indicating the exact dependency of the bound on the variable. In order to achieve that in practice, we developed a tool, RESANA[†], that contains a general process which has two phases.

[†]RESANA is open source software and can be downloaded from http://resourceanalysis.cs.ru.nl/resana/.

Inference In the inference phase the RESANA tool analyses the JAVA source of the program in order to propose a possible resource bound for the program. It uses traditional analysis techniques like solving cost-relation systems and a novel polynomial interpolation technique. This interpolation-based approach is very powerful. It allows also non-monotonic polynomial bounds to be derived (the developer does not have to indicate the exact dependencies: they are derived). The obtained result is added to the JAVA program via an annotation using the JML specification language [11].

Verification Results are achieved by solving cost relations or by interpolating polynomials. Solving cost relations is sound by construction. The use of interpolation is not guaranteed to be sound. Therefore, the results achieved by interpolation must be verified, e.g. by the KEY verification tool [12] or the QEPCAD algebraic decomposition tool [13]. If the tool is not able to verify them, one can proceed with a new inference phase with other user options, such as e.g. trying a higher degree polynomial.

The RESANA tool supports three kinds of analysis.

Loop Bound Analysis An expression that gives a symbolic upper bound for the number times a loop is executed may be derived and verified using the integrated combination [14] of the tools RESANA and KEY.

Heap Bound Analysis An expression for a symbolic upper bound of the consumed heap is derived using RESANA extended with a variant of the external tool COSTA [15]. The COSTA tool has been adapted to produce accurate values for OPENJDK, as well as the real-time JAMAICAVM virtual machine [16]. Furthermore, the capabilities of the COSTA tool have been enlarged through the internal use of interpolation technology [17].

Stack Bound Analysis An expression for a symbolic upper bound of the space for the stack is derived using RESANA with the enlarged COSTA that provides an upper bound for the depth of recursive calls; this information is used by the VERIFLUX tool [18] to obtain a *numeric* stack bound.

These three kinds of analysis are integrated in a common program development environment through an ECLIPSE plug-in, such that a developer can easily switch between development and verification activities guaranteeing the memory safety of critical real-time software applications.

In Section 2 loop bound analysis is described. Section 3 presents heap bound analysis and the adjustments that have been made to make it applicable in practice. Analysing stack bounds is discussed in Section 4. User experience with RESANA is described in Section 5. Finally, in Sections 6 and 7, related work is discussed and conclusions are drawn.

2. LOOP-BOUND ANALYSIS

In order to prove the termination of a piece of software or, even harder, to calculate bounds on run-time or usage of resources such as heap space or energy, finding bounds on the number of iterations that the loops can make is a prerequisite. While in some cases a loop may iterate a fixed number of times, its execution will often depend on program input. Therefore we consider *symbolic* loop bounds, or *ranking functions*.

A loop ranking function is a function over (some of) the program variables used in the loop, that decreases at each iteration and is bounded by zero. Listing 1 shows a simple while loop. Although $100 - i$ is a perfectly fine ranking function as well, the most precise one for this loop is $15 - i$. This gives the exact number of iterations the loop will make, for arbitrary i (given that $i < 15$, see Section 2.6).

```
while (i < 15)
  i++;
```

Listing 1: A simple while loop, with most precise ranking function $15 - i$.

In this section, we present a method for the automatic inference of polynomial ranking functions for loops, based on polynomial interpolation. The basic procedure was first presented by Shkaravska et al. in [14]. It can infer *polynomial* ranking functions, whereas other methods are limited to linear symbolic or concrete bounds. Note that to derive concrete bounds from symbolic bounds, the analysis could be combined with data-flow analysis. To derive concrete upper and lower bounds on the number of iterations of a loop, upper and lower bounds have to be known statically for all the program variables in the symbolic bound.

We introduce polynomial-interpolation-based ranking function inference in Section 2.1. In Section 2.2, a quadratic example is given. The soundness of the method is discussed in Section 2.3. Then, extensions to the basic method are discussed in Section 2.4 (ranking functions with rational or real coefficients), Section 2.5 (branching inside the loop body) and Section 2.6 (disjunctional loop guards). In Section 2.7 a limitation to the extension for disjunctional loop guards and a solution are discussed. Another application of our polynomial interpolation method is discussed in Section 3.1.

2.1. *Test-Based Inference of Polynomial Ranking Functions for Loops*

In [14], Shkaravska, Kersten and Van Eekelen present a method for the inference of polynomial ranking functions for loops. Only loops in which the guards are conjunctions over arithmetical (in)equalities are considered:

$$guard := inequality \mid inequality \wedge guard$$

$$inequality := num_1 \ \mathbf{b} \ num_2$$

where num_i is a numerical program variable or constant and $\mathbf{b} := \{<, >, =, \neq, \leq, \geq\}$.
The method proceeds in the following steps:

1. Instrument the loop with a counter
2. Run tests on a *well-chosen* set of input values
3. Find *the* polynomial interpolation of the results

Here, *well-chosen* means that test-nodes have to be picked such that there exists a unique interpolating polynomial. This is the reason we can refer to *the* polynomial interpolation in step 3. Remember that a polynomial $p(z_1, \ldots, z_k)$ of degree d and dimension k (the number of variables) has $N_d^k = \binom{d+k}{k} = \frac{(d+k)!}{d! \cdot k!}$ coefficients. This is the number of test-nodes that we need. To ensure the existence of a unique interpolation, the test nodes are chosen to lie in so-called Node Configuration A (NCA). This condition was first presented in [19]; its application to loop-bound analysis is described in [14]. Besides lying in NCA, test-nodes must also satisfy the guard of the considered loop. An algorithm for node search is presented in [14].

In the current version of RESANA, the ranking function can contain primitive data types, object field access and array access. Note that in theory, the method could also handle loops for which the ranking function depends on for instance the height of a tree. However, since this height is not readily available in a program variable, this would require the addition of such a variable expressing the tree height by the programmer.

2.2. *Quadratic Example*

Consider the example in Listing 2. The most precise ranking function for this loop is the degree 2 polynomial $a \cdot b - c + 1$.

```
while (a > 0 && c <= b && c > 0) {
  if (c == b) { a--; c = 0; }
  c++;
}
```

Listing 2: A while loop with degree 2 ranking function $a \cdot b - c + 1$.

The inference of a ranking function for the loop in Listing 2 is depicted in Figure 1. First, the loop is instrumented with a counter. The user inputs the expected degree 2 of the polynomial ranking

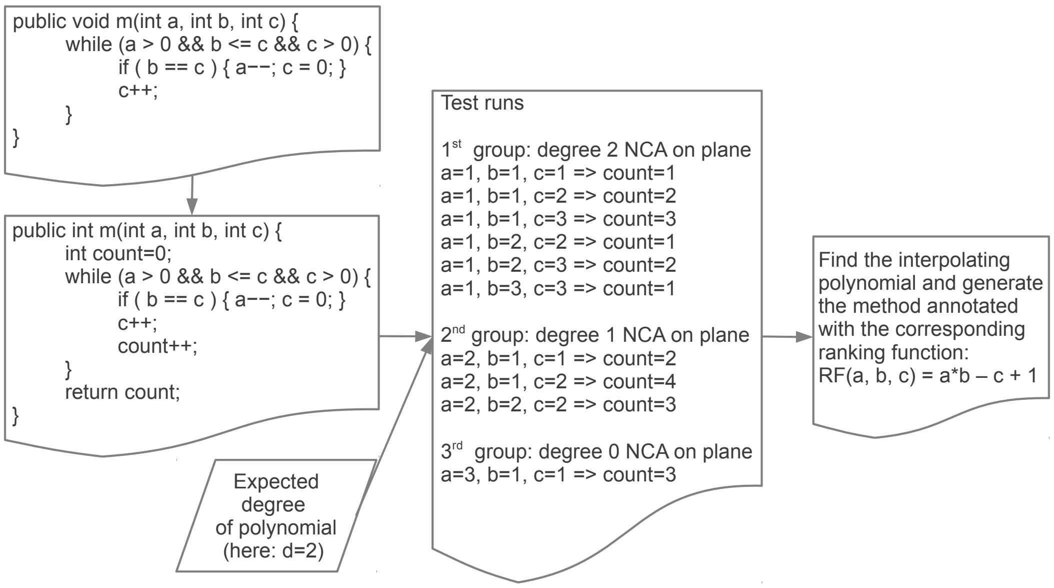
public void m(int a, int b, int c) {
while (a > 0 && b <= c && c > 0) {
if (b == c) { a−−; c = 0; }
c++;
}
}
public int m(int a, int b, int c) {
int count=0;
while (a > 0 && b <= c && c > 0) {
if (b == c) { a−−; c = 0; }
c++;
count++;
}
return count;
}
Expected degree of polynomial (here: d=2)
Test runs
1st group: degree 2 NCA on plane
a=1, b=1, c=1 => count=1
a=1, b=1, c=2 => count=2
a=1, b=1, c=3 => count=3
a=1, b=2, c=2 => count=1
a=1, b=2, c=3 => count=2
a=1, b=3, c=3 => count=1
2nd group: degree 1 NCA on plane
a=2, b=1, c=1 => count=2
a=2, b=1, c=2 => count=4
a=2, b=2, c=2 => count=3
3rd group: degree 0 NCA on plane
a=3, b=1, c=1 => count=3
Find the interpolating polynomial and generate the method annotated with the corresponding ranking function:
RF(a, b, c) = a*b – c + 1

Figure 1. Test-based inference method applied to the example from Listing 2.

function. Since there are 3 variables, a set of $N_2^3 = \frac{(2+3)!}{2!\cdot 3!} = 10$ test-nodes in NCA is generated. By interpolating the results from test runs using these input values, the most precise quadratic ranking function $a \cdot b - c + 1$ is found.

2.3. Soundness

The presented method infers a *hypothetical* ranking function. It is not sound by itself, but requires an external verifier. The Java Modelling Language (JML) is used to express the ranking functions [20]. Inferred ranking functions are expressed in JML by defining a `decreases` clause on the loop. This is an expression which must decrease by at least 1 on each iteration and has a value greater than or equal to 0, see the JML reference manual [11]. It therefore forms an upper-bound on the number of iterations of the loop. An example is shown in Listing 3.

When the loop condition does not hold, the loop iterates zero times. Therefore the shown annotation actually expresses the maximum of $a \cdot b - c + 1$ and 0. In general, a ranking function $RF(\bar{v})$ for a loop with condition b can be expressed as follows: `decreases` b ? $RF(\bar{v})$: 0. Such JML annotations can be verified by a variety of tools, for instance KEY [12]. The procedure described here should be used in conjunction with such a prover to provide soundness.

```
//@ decreases (a > 0) && (c <= b) && (c > 0) ? a * b − c + 1 : 0;
while (a > 0 && c <= b && c > 0) {
  if (c == b) { a−−; c = 0; }
  c++;
}
```

Listing 3: The loop from Listing 2, annotated with its ranking function

Figure 2 depicts a bird's eye view of the overall procedure. After a ranking function is inferred, the Java sources are annotated and sent to the verification tool (KEY). The verifier might be able to prove correctness of the annotation automatically, manual steps may be needed for complex ranking functions (non-linear, rational coefficients, et cetera) or the user may not be able to construct a proof at all. In the latter case, the user can go back and try the procedure for a higher expected degree of the polynomial ranking function. If an expected degree higher than the actual degree of the polynomial is used, the correct result will still be found. There will however be a performance penalty on the analysis.

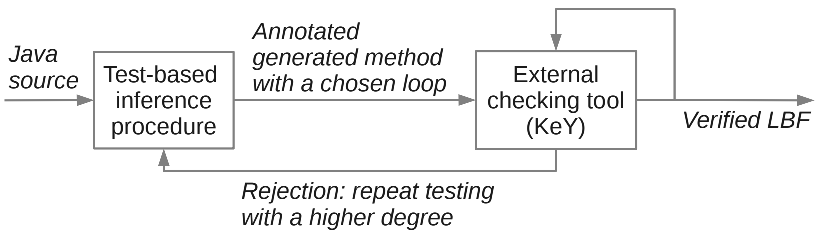
Java source
Test-based inference procedure
Annotated generated method with a chosen loop
External checking tool (KeY)
Verified LBF
Rejection: repeat testing with a higher degree

Figure 2. The basic inference procedure from a bird's eye view: infer-and-check cycle.

2.4. *Extension: Ranking Functions with Rational or Real Coefficients*

The ranking functions inferred by the basic method are polynomials with coefficients that are natural, rational or real numbers. However, when a polynomial has rational or real coefficients, its result is not necessarily a natural number, which, of course, any estimate of a number of loop iterations must be. Consider for instance the loop in Listing 4.

```
while (start < end) {
  start += 4;
}
```

Listing 4: An example with a loop-bound function that is a polynomial over rational coefficients

The exact number of iterations of this loop is given by $\lceil\frac{\texttt{end}-\texttt{start}}{4}\rceil$. In other words, when $\frac{\texttt{end}-\texttt{start}}{4}$ does not equate to a natural number, for instance to $\frac{3}{4}$, it must be *ceiled*. In general, when the coefficients of an inferred polynomial ranking function $RF(\bar{v})$ are not natural numbers, ceiling should be added as such: $\lceil RF(\bar{v})\rceil$. Unfortunately, there is no ceiling operator in JML. KEY simply truncates non-integer values after the decimal. We therefore chose to overestimate ceiling by adding one to the KEY truncation: $\lceil RF(\bar{v})\rceil \leq RF(\bar{v}) + 1$.

When choosing test nodes for the loop in Listing 4 naively, for instance (0,1), (1,2) and (1,3), an incorrect ranking function will be the result (in this case the constant 1). We must take into account that if a variable v is updated by increasing or decreasing by a constant $step$, the test-nodes must lie $step$ apart. In this example, if we pick test nodes (0,4), (4,8) and (4,12), then the correct ranking function will be found.

2.5. *Extension: Branching inside the loop body*

The basic procedure finds correct ranking functions for most loops containing branching, such as for example the one in Listing 2. However, there are cases in which the basic procedure fails, because the different paths affect the bound in different ways. Such a case is shown in Listing 5.

```
while (i > 0)
  if (i > 100) i −= 10;
  else i −= 1;
```

Listing 5: Example where the basic method supplies an incorrect ranking function. Therefore, branch-splitting is applied, yielding the pessimistic, but correct ranking function i.

To solve this problem, we have invented *branch-splitting*. This procedure finds ranking functions for loops where the if-statements, if they exist in a loop body, have the following *worst-case computation path (WCCP) property*:

For each loop body, there is an execution path such that, for any collections of values of the loop variables, if one follows this execution path in every loop iteration one reaches the worst-case, i.e. the upper bound on the number of iterations.

This condition is not checked by the loop bound inference procedure. It is given here to specify the class of loops for which the procedure is successful. Soundness of the result is ensured by verification using KEY.

With branch-splitting, we mean that we generate multiple new loops from the original, one for each possible path. We then do the analysis for each of these paths. The ranking function is then the maximum of all the inferred ranking functions. Thanks to the WCCP property, we can easily find the ranking function that always specifies the maximum, by supplying a set of values for the variables (say, all ones) to all the ranking functions. For the example in Listing 5, this yields the ranking function i.

2.6. *Extension: Piecewise Ranking Functions for Loops with Disjunctive Guards*

In this section, we formally describe an extension to the basic procedure for handling loops with disjunctions in their guards. The set of considered loops is here thus extended to those with as guard *any* propositional logical expression over arithmetical (in)equalities, including disjunctions. We will see that for those loops for which the guard contains disjunctions, the ranking function will become *piecewise*.

Note that in fact, any ranking function for a well-formed loop is a piecewise one, since there is always the piece where the loop guard does not hold and the loop iterates zero times. For instance, for the loop in Listing 1, the ranking function is actually:

$$\begin{cases} \texttt{15} - \texttt{i} & \text{if } (\texttt{i} < \texttt{15}) \\ 0 & \text{else} \end{cases} \tag{1}$$

This is of course a trivial case. A more involved example of a loop for which a piecewise ranking function can be defined is shown in Listing 6.

```
while ((i>0 && i<20) || i>50) {
  if (i>50) i--;
  else i++;
}
```

Listing 6: While loop with a piecewise ranking function.

Its ranking function is the following:

$$\begin{cases} \texttt{20} - \texttt{i} & \text{if } (\texttt{i} > \texttt{0}) \wedge (\texttt{i} < \texttt{20}) \\ \texttt{i} - \texttt{50} & \text{if } \texttt{i} > \texttt{50} \\ 0 & \text{else} \end{cases} \tag{2}$$

We will now formally define a generic method for inferring ranking functions for loops with disjunctive guards. The first step is to transform the guard into disjunctive normal form (DNF), using the laws of distribution and DeMorgan's theorems. Thereafter it has the form:

$$guard := conj \mid conj \vee guard$$

$$conj := inequality \mid inequality \wedge conj$$

$$inequality := num_1 \, \mathbf{b} \, num_2$$

where num_i is a numerical program variable or constant and $\mathbf{b} := \{<, >, =, \neq, \leq, \geq\}$.

Each conjunction c_i represents a logical conjunction over numerical (in)equalities. We can now split up the guard by applying the following function:

$$DNFsplit(c_1 \vee \ldots \vee c_n) := \left\{ \bigwedge_{c_i \in CP} c_i \wedge \bigwedge_{c_j \in C_{rest}} \neg c_j \;\middle|\; \begin{array}{l} CP \in \mathcal{P}(\{c_1, \ldots, c_n\}) \backslash \emptyset \\ C_{rest} = \{c_1, \ldots, c_n\} \backslash CP \end{array} \right\}$$

This transforms the condition $c_1 \vee \ldots \vee c_n$ into a set $Pieces$ of $2^n - 1$ conjunctive conditions. For instance, $DNFsplit(i > 10 \vee i < 3)$ yields three pieces: $i > 10 \wedge \neg i < 3$, $i < 3 \wedge \neg i > 10$ and

$i > 10 \wedge i < 3$. This set may be simplified using a Satisfiability Modulo Theories (SMT) solver. In this case, the negations can be removed from the first two conditions. The third condition is unsatisfiable, thus it may be removed altogether. We refer to the procedure of transforming a guard into disjunctive normal form and separating the pieces as *DNF-splitting*. The set $Pieces$ defines the pieces of the piecewise polynomial ranking function.

After DNF-splitting, the basic method can be applied separately for each of the pieces. If RF_p is the polynomial ranking function inferred for a piece $p \in Pieces$, then this yields the following piecewise ranking function:

$$\begin{cases} RF_{p_1} & \text{if } p_1 \\ \ldots & \text{if} \ldots \\ RF_{p_m} & \text{if } p_m \\ 0 & \text{else} \end{cases} \tag{3}$$

In this piecewise polynomial ranking function, $m \leq 2^n - 1$, because unsatisfiable pieces have been removed.

2.7. *Condition Jumping*

In this section we define a complication that may arise during DNF-splitting, which we call *condition jumping*. We show how to detect its occurrence and how to infer ranking functions even in the presence of condition jumping.

```
while ((i>0 && i<20) || i>22) {
  if (i>22) i--;
  else i+=4;
}
```

Listing 7: While loop with jumping between the disjunctive conditions.

Consider the loop in Listing 7. Naively, one could say that its ranking function is the following:

$$\begin{cases} \lceil (20 - \mathtt{i})/4 \rceil & \text{if } (\mathtt{i} > 0) \wedge (\mathtt{i} < 20) \\ \mathtt{i} - 22 & \text{if } \mathtt{i} > 22 \\ 0 & \text{else} \end{cases} \tag{4}$$

But, what if $\mathtt{i}$ is 19, 15, or any $n \in [1, 19]$ with $n \bmod 4 = 3$? Indeed, then there is a shift from the first condition ($0 < i < 20$) to the second one ($i > 22$). We call this *condition jumping*. Jumping from the second condition into the first one is not possible in this case.

Because of the presence of condition jumping, regular DNF-splitting does not suffice here. The set of nodes from which condition jumping occurs must be considered as a separate piece, as follows:

$$\begin{cases} \lceil (20 - \mathtt{i})/4 \rceil + 1 & \text{if } (\mathtt{i} > 0) \wedge (\mathtt{i} < 20) \wedge i \bmod 4 = 3 \\ \lceil (20 - \mathtt{i})/4 \rceil & \text{if } (\mathtt{i} > 0) \wedge (\mathtt{i} < 20) \wedge i \bmod 4 \neq 3 \\ \mathtt{i} - 22 & \text{if } \mathtt{i} > 22 \\ 0 & \text{else} \end{cases} \tag{5}$$

A method to detect condition jumping is described in Section 2.7.1. This method is then applied in an algorithm which detects all nodes for which jumping occurs in Section 2.7.2, in order to infer a correct piecewise ranking function.

2.7.1. Detection of Condition Jumping using Symbolic Execution and SMT Solving To detect condition jumping in the example in Listing 7, we first use symbolic execution [21] to construct an update function, which captures the relation between the values of the program variables pre and post execution of the loop body. We can then use this relation as input to an SMT solver and search for a model for which one part of the loop guard is true pre-execution of the loop body and another part is true post-execution.

Obtaining an update function We will name the pre/post execution relation for a variable v the $next_v$ function. The function $next_i :: Int \rightarrow Int$ for the loop in Listing 7 can be determined by symbolically executing the loop with value α_i for i. This results in the following symbolic post-execution value, which we will name ϕ_i:

$$\phi_i(\alpha_i) = \begin{cases} \alpha_\mathtt{i} - 1 & \text{if } \alpha_\mathtt{i} > 22 \\ \alpha_\mathtt{i} + 4 & \text{if } \neg(\alpha_\mathtt{i} > 22) \end{cases} \tag{6}$$

By replacing the α symbol by i, this easily translates to the $next_i$ function we were looking for:

$$next_i(i) = \begin{cases} \mathtt{i} - 1 & \text{if } \mathtt{i} > 22 \\ \mathtt{i} + 4 & \text{if } \neg(\mathtt{i} > 22) \end{cases} \tag{7}$$

In general, such an update function can be derived by symbolic execution of the loop body. Start by giving the variables $v_1 \ldots v_n$ symbolic values $\alpha_1, \ldots, \alpha_n$. After the symbolic execution of the loop body, each variable v_i will now have a value which is a set of polynomials over the symbols $\alpha_1, \ldots, \alpha_n$ and constants, with associated *path conditions*, which capture branching. Effectively, this is again a piecewise polynomial. The function $next_{v_i}$ is now obtained by replacing the α's by the corresponding program variables in this piecewise polynomial.

Detecting condition jumping The SMT-LIB is a library of SMT background theories and benchmarks [22]. It has a common file format for SMT problems, which can be read by most SMT-solvers. An SMT-LIB script to detect jumping in the example from Listing 7 is given in Listing 8. The function $next_i :: Int \rightarrow Int$ from Equation 7 is defined on line 2. Then on line 4 we define the condition expressing that jumping occurs for this example and on line 6 we check satisfiability of this condition.

```
1 (declare-fun i () Int)
2 (define-fun nexti ((x Int)) Int
3   (ite (> x 22) (- x 1) (+ x 4)))
4 (assert (and (and (> i 0) (< i 20))
5   (> (nexti i) 22)))
6 (check-sat)
7 (exit)
```

Listing 8: SMT-LIB script to detect jumping in the code of Listing 7.

Let us now consider the general case. Condition jumping will be detected pairwise for conditions with multiple disjunctions. Here we thus consider a single condition-pair, i.e. a loop with guard $b_1 \vee b_2$. Here b_1 and b_2 are conditions over $CV \subseteq LV \subseteq PV$, where CV are the program variables in the condition, LV are the program variables in the loop and PV are all program variables.

For each $v_i \in LV$, we can define an associated function $next_{v_i} :: T_{v_1} \rightarrow \ldots \rightarrow T_{v_i} \rightarrow \ldots \rightarrow T_{v_n} \rightarrow T_{v_i}$, where T_{v_i} is the type of v_i and $n = |LV|$, which takes the values of all $v \in LV$ as the state and computes the value of v after a single execution of the loop body in that state, by following the procedure described in the previous paragraph. Once these functions have been derived, the question whether jumping from b_1 to b_2 is possible can be answered by any SMT-LIB conforming SMT-solver[‡] by determining the satisfiability of $b_1(v_1, \ldots, v_n) \wedge b_2(next_{v_1}(LV), \ldots, next_{v_n}(LV))$.

2.7.2. *Generating Ranking Functions in the Presence of Condition Jumping*

The SMT-LIB script in Listing 8 can be used to find a *model* for which jumping occurs by adding the expression `(get-value (i))`. A model is an instantiation of the variables for which the formula for which satisfiability is checked holds. In the SMT-LIB script from Listing 8, a model for i is 19.

[‡]For instance Z3, which can be used on-line at http://research.microsoft.com/en-us/um/redmond/projects/z3/

Subsequently, by adding the expression `(assert (distinct i 19))`, one can search for models *other than* $i = 19$ for which jumping occurs. The answer of the SMT solver is that the combination of propositions in this script is unsatisfiable. Thus, $i = 19$ is the only possible model. We can now see if there are any models from which the state $i = 19$ can be reached in a single iteration, by changing line 6 to `(= ( nexti i) 19)))`. In the example, this will be the model $i = 15$. Subsequently and similarly, we can search for other nodes that can reach the state $i = 19$ in a single step, or that can reach the state $i = 15$. By repeating these steps, we can find the set $J = \{3, 7, 11, 15, 19\}$. These are the models from which jumping can occur.

In general, the method described in Section 2.7.1 can be extended to detect all models from which condition jumping can occur, by first finding all models that can jump directly from b_1 to b_2 and then recursively finding models that can reach a model from this first set. This can be done by implementing the following algorithm around an SMT-solver. In this algorithm, J is the set of models of which it is known that condition jumping occurs and Q is a queue of models. We assume a function $next :: M \rightarrow M$ (where M is the type of a model), which applies to each variable v_i in a model $\bar{v}$ its corresponding $next_{v_i}$ function.

1. Is there a model $\bar{v}$ for which $b_1(\bar{v}) \wedge b_2(next(\bar{v})) \wedge \bar{v} \notin J$?
   - SAT $\rightarrow$ Add $\bar{v}$ to J and Q, goto 1.
   - UNSAT $\rightarrow$ Goto 2.
2. Q empty?
   - Yes $\rightarrow$ Done.
   - No $\rightarrow$ Goto 3.
3. For a model $\bar{q}$ on the queue Q, is there a model $\bar{v}$ for which $b_1(\bar{v}) \wedge next(\bar{v}) = \bar{q} \wedge \bar{v} \notin J$?
   - SAT $\rightarrow$ Add $\bar{v}$ to J and Q, goto 3.
   - UNSAT $\rightarrow$ Remove $\bar{q}$ from Q, goto 2.

After execution, J contains exactly all nodes for which jumping occurs. Since here a queue is used, this algorithm implements a breadth-first search. This can easily be adapted to a depth-first search by using a stack. Since the set of models is finite, the algorithm will always terminate. It may however require $|J|$ runs of the SMT-solver, so one may choose to set an upper bound on the size of J and abort ("give up") early.

Now that we know J, we can split condition b_1 into two: $b_1(\bar{v}) \wedge \bar{v} \in J$ and $b_1(\bar{v}) \wedge \bar{v} \notin J$. We can then apply the basic method to each of these disjunctive pieces. This algorithm only detects jumping from one piece into another. It should be applied iteratively over all the pieces, until no more jumping can occur. Note that this approach does not terminate until all condition jumping cases have been found. Since there are loops for which jumping occurs for every value of for instance an Integer, it should "give up" after an upper-bound on the number of jumps is reached.

3. HEAP-SPACE USAGE ANALYSIS

RESANA's heap consumption analysis is based on the COSTA [15] tool, which provides a generic analysis infrastructure for JAVA byte code. The symbolic upper bound that COSTA generates for a method depends on the logical sizes of the method's arguments, structures pointed to by the object fields and the costs of the called (library) methods. The (logical) size of an integer is the maximum of the integer and 0, the size of an array is its length, the size of an object is its maximal reference chain. These assumptions constitute the size model in the COSTA terminology. For instance, let a method allocate n objects of class `X`, where integer n is a parameter of the method. Then COSTA generates a symbolic bound of the form $nat(n) * c(size(\texttt{X}))$, where $nat(n)$ is integer n its logical size: $max\{n, 0\}$ and $c(size(\texttt{X}))$ is the memory cost of creating an object of type `X`.

COSTA implements different garbage collection models [23]. This functionality is retained in RESANA. Inside JAVA real-time threads no garbage collection is used, so in RESANA a user can select to ignore garbage collection. For normal JAVA code one can select to use the garbage collection feature of COSTA, which calculates an upper bound for all possible executions of a program. First, for every method, the amount of memory that can escape the method's scope is deduced. Using this information, peak consumption cost relationships are calculated and solved, which give upper bounds on the amount of memory used, even if using garbage collection.

We have added a number of improvements to the existing COSTA tool. The recurrence solver was improved with interpolation-based height analysis. Secondly, the ability to calculate concrete bounds for a number of JAVA Virtual Machines, like OPENJDK and JAMAICAVM, was added. Third, we changed the bounds calculation of arrays, from an under-approximation to a over-approximation. And finally we added a post-processing step to simplify the expressions, so a programmer can easily interpret the information.

3.1. Interpolation-based height analysis for improving a recurrence solver

COSTA's approach to resource analysis is based on the classical method devised by Wegbreit [24], which involves the generation of a recurrence relation capturing the costs of the program being analysed, and the computation of a closed form (non-recursive cost expression) which bounds the results of this recurrence relation. In COSTA terminology, recurrence relations are called *Cost Relation Systems* (CRS). The main feature that distinguishes CRSs from the classical concept of recurrence relations is non-determinism: a CRS defining the costs of a JAVA method may be defined by a set of equations guarded by non-disjoint conditions. As an example, consider the loop in Listing 9.

```
while (x <= y) {
  new Object();
  if (...) x = x + 1; else y = y - 2;
}
```

Listing 9: Example loop.

We assume that the value of the `if` condition cannot be determined at compile time. Its memory costs are described by the following (simplified) CRS:

$$T(x,y) = 0 \qquad \{x > y\} \tag{8}$$
$$T(x,y) = c + T(x+1,y) \qquad \{x \leq y\} \tag{9}$$
$$T(x,y) = c + T(x,y-2) \qquad \{x \leq y\} \tag{10}$$

where c denotes the constant $c(\mathit{size}(\texttt{java.lang.Object}))$, i.e. the memory cost of creating an instance of `Object`. The COSTA system provides the recurrence solver PUBS [4], which computes the following closed-form:

$$\begin{gathered} nat(y-x+1) * c(\mathit{size}(\texttt{java.lang.Object})) \\ + c(\mathit{size}(\texttt{java.lang.Object})) \end{gathered}$$

This is an upper-bound to the values of $T(x,y)$ given above.The resulting closed form corresponds to the worst-case execution of the loop (i.e. when the `if` condition always holds).

An important issue in the search of a closed-form of a CRS is to approximate the maximum number of unfoldings that must be undergone in order to reach a base case (height analysis). If we consider the CRS as a function being evaluated in a non-deterministic way, the number of unfoldings is closely related with the concept of ranking functions (see Section 2). For instance, in the CRS given above we get the following unfolding sequence of length $y-x+1$:

$$\underbrace{T(x,y) \rightarrow T(x+1,y) \rightarrow T(x+2,y) \rightarrow \cdots \rightarrow T(y,y)}_{y-x+1 \text{ unfoldings}}$$

PUBS derives a ranking function for T by applying Podelski and Rybalchenko's method [25], which is complete for linear ranking functions. Unfortunately, it fails when the number of unfoldings does not depend linearly on the arguments of the CRS, as the following example shows:

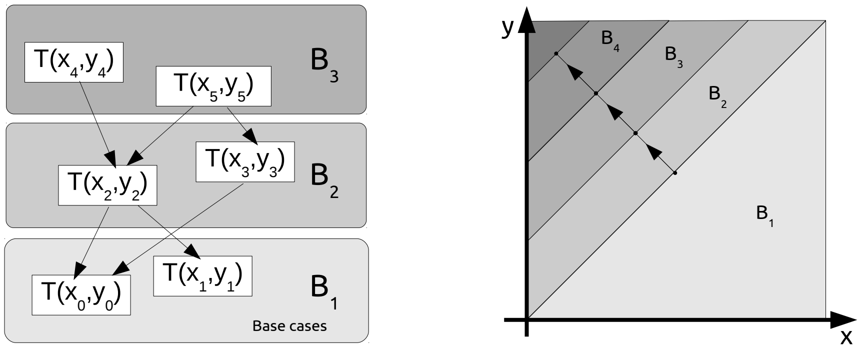

Figure 3. Meaning of the B_i sets and their representation as convex polyhedra.

$$R(x, y) = c \qquad \{x = 0, y = 0\} \tag{11}$$
$$R(x, y) = c + R(x - 1, x - 1) \qquad \{x > 0, y = 0\} \tag{12}$$
$$R(x, y) = c + R(x, y - 1) \qquad \{x \geq 0, y > 0\} \tag{13}$$

By equation (13) the variable y is decreased in every recursive call, until it reaches zero. Then, by equation (12) it is set to $x - 1$, from which it starts decreasing again. The worst-case evaluation of $R(x, y)$ yields a chain of length $\frac{1}{2}x^2 + \frac{1}{2}x + y + 1$, which does not depend linearly on (x, y).

We have extended the PUBS system so that it can infer polynomial ranking functions via testing and polynomial interpolation, as has been explained in Section 2. This extension was described in detail in [17]. It is described briefly here, with an additional contribution of verification of the interpolation results. The approach is, essentially, the same: choose a set of points (lying in a NCA) in the domain of the relation defined by the CRS, evaluate the CRS at these points, and find the interpolating polynomial. However, the evaluation of a CRS is more involved than the evaluation of a program instrumented with a counter, as it was done in Section 2.1. The main difficulty lies in non-determinism. Assume we want to evaluate $T(5, 9)$, where T is defined as in the CRS shown in (8-10). We can unfold the definition of $T(5, 9)$ by using (10) until we reach a base case, so we get the following sequence:

$$T(5, 9) \rightarrow T(5, 7) \rightarrow T(5, 5)$$

This sequence is of length three, which is not maximal, since we could have evaluated T by always using (9), so as to obtain a longer sequence:

$$T(5, 9) \rightarrow T(6, 9) \rightarrow T(7, 9) \rightarrow T(8, 9) \rightarrow T(9, 9)$$

As a consequence of this, we would have to examine all the possible evaluations of $T(5, 9)$ in order to obtain the longest unfolding sequence. However, the number of possible evaluations may be infinite even if the evaluation yields a finite number of results. We have addressed this problem by evaluating the CRS in a bottom-up way (Figure 3 left). We start from the set B_1 of pairs (x, y) such that the evaluation of $T(x, y)$ does not fall into a recursive case. The longest obtainable sequence in these cases is of length one. Now let us define the set B_2 of pairs (x, y) such that the evaluation of $T(x, y)$ falls into a recursive case, but the recursive call belongs to B_1. Thus we ensure that the evaluation of these pairs does not require more than two unfoldings. By following this procedure we obtain a sequence of sets $\{B_i\}$ each of which can be described as a disjoint union of convex polyhedra with the help of quantifier elimination techniques. We use a gradient-based approach for selecting the interpolation nodes from the B_i sets (Figure 3 right). The algorithm involves the search of *climbing paths* starting at the B_1 set, and minimizing the distance between B_i and B_{i+1} for each $i \in \mathbb{N}$. It is possible that, given a point (x, y) in a set B_i, there are several candidates in the next level B_{i+1} lying at the same distance from (x, y). In this case the climbing path forks, and the next interpolation nodes are searched from all these candidates. The process ends when the interpolating polynomial is uniquely determined.

Once we have found the interpolating polynomial on the set of test nodes, we have to check whether the resulting bound is correct. This can be done as follows: for each CRS the system can derive some predicates, whose satisfiability is a sufficient condition guaranteeing that the polynomial is an upper bound to the values of the CRS. These conditions involve inequalities between polynomial expressions, which are decidable in Tarski's theory of real closed fields. For instance, the system would generate the following logical statement for checking that $y - x + 1$ is an upper bound to $T(x, y)$:

$$\forall x, y, x', y' : ((x \leq y \wedge x' = x + 1 \wedge y' = y) \vee (x \leq y \wedge x' = x \wedge y' = y - 2)) \\ \implies y - x + 1 \geq 1 + y' - x' + 1$$

If these generated predicates hold, then $y - x + 1$ is indeed an upper bound to $T(x, y)$. Our extension to PUBS delegates the task of checking such inequalities to the QEPCAD tool [13]. For instance, in our running example $T(x, y)$ the following script is generated[§]:

```
[ Proving correctness of the bound corresponding to simpleLoop ]
(x,y,x',y')    -- Variables
0              -- Number of free variables in the formula
(A x) (A y) (A x') (A y')
  [[[x >= 0 /\ y >= 0 /\ x' >= 0 /\ y' >= 0] /\
  [[[(-1) x + 1 y' >= (-2) /\ 1 x + (-1) x' = 0 /\ 1 y + (-1) y' = 2]  \/
    [(-1) x + 1 y' >= 0 /\ 1 x + (-1) x' = (-1) /\ 1 y + (-1) y' = 0]]]]
     ==> [(-1) x + 1 y + 1 >= 1 + (-1) x' + 1 y' + 1]].
finish
```

Given this script, QEPCAD yields the message `An equivalent quantifier-free formula: TRUE`, which validates the inferred bound.

3.2. *Correct array-size analysis*

Due to the way memory is handled, an array header will always be included with information about the array. As an array is a regular JAVA object the array header also includes the normal object header. Almost all architectures impose constraints on the memory allocator, e.g. memory allocators on the x86 architecture will allocate memory blocks in multiples of quad word sizes, so in multiples of 16 bytes. Although less bytes are requested, the memory allocator will add padding to an object that cannot be used for other purposes. This array header and padding need to be taken into account, otherwise the bound would be an under-approximation.

For instance, all JAMAICAVM allocations are in (multiple) blocks of 32 bytes, considering the 32-bit version of JAMAICAVM. If multiple blocks are needed they are stored in a tree structure with the array content stored in the leafs of the tree. The array header is 16 bytes long, so this leaves up to four pointers to the tree structures. In partial trees (in which the number of elements is not 4×8^n), nodes leading to unused array contents and unused array contents blocks are not stored, e.g. 16 pointers (four bytes each) stored will take only three blocks: two for the leafs and one intermediate block pointing to the leafs [16]. An example array structure is shown in Figure 4. COSTA takes into account neither the array header, nor the structure needed to store the contents, nor padding. Only the space needed by the array contents (object references and primitive types) is included in the bound. This results in COSTA producing a bound for `new int[n]` equal to $n * size(\texttt{int})$, making it indistinguishable from the sequence `new int[1]; new int[n-1];`, so neglecting to account for the extra array header, padding and structure overhead. The (structure) overhead is dependent on the virtual machine used. To deal with these deficiencies we implemented a special mode in COSTA when generating a concrete bound for arrays in JAMAICAVM, as explained next.

[§]Variables have been renamed for better readability.

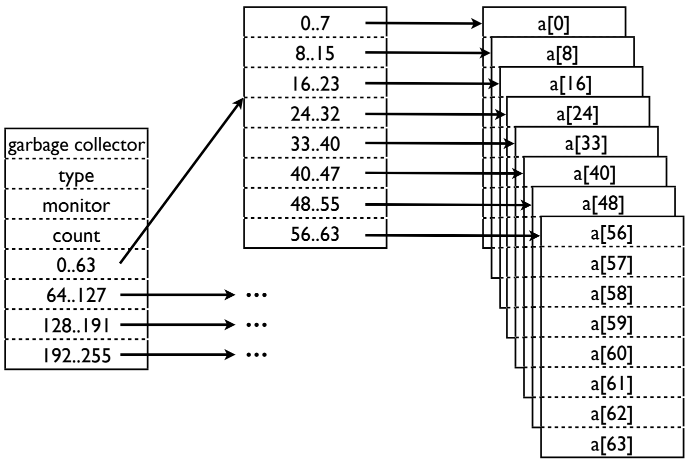

Figure 4. Graphical representation of a JAMAICAVM array of size n, with $33 \leq n \leq 255$, with $a[i]$ representing the contents of the array. Allocating an array of 63 elements takes 10 blocks.

3.3. Virtual-machine specialisation by adding type-size information

COSTA has no knowledge of specific JAVA Virtual Machines like JAMAICAVM. Our approach is to replace in all the symbolic bounds generated by COSTA the symbolic object sizes by the exact sizes of objects in bytes. The exact sizes are retrieved from the target VM by means of a specially generated program. For JAMAICAVM, this generated program depends on the Scoped Memory extensions of REALTIME JAVA. For other JAVA VMs we use the reflection API in JAVA, which is more general and can be run on any VM which supports the reflection API. We validated this method for OPENJDK, by interfacing directly with the virtual machine by means of a (JNI) plugin.

For generating bounds for arrays allocated in an instance of JAMAICAVM, we adjusted COSTA to include an over-approximation. A simple way of calculating the size of arrays, by means of the small recursive function defined in Equation 14, could not be implemented in COSTA, because of the manner COSTA represents and calculates the bounds internally. This recursive function is valid for data types of four bytes¶, which correspond to the size of pointers used in the tree structure pointing to the leafs, resulting in a cleaner formula.

$$\text{arrayblocks}(n) \quad = \quad \begin{cases} n & \text{if } n \leq 8 \\ \lceil \frac{n}{8} \rceil + \text{arrayblocks}(\lceil \frac{n}{8} \rceil) & \text{otherwise} \end{cases} \tag{14}$$

By transforming the formula to an over-approximation (by replacing $\lceil \frac{n}{8} \rceil$ with $\frac{n+7}{8}$), we were able to solve this new recurrence equation. The results in a new formula, and after integrating adjustments for the start cases, is listed in Equation 15. We have implemented this solution in our version of COSTA, which is included in RESANA, so that analysing arrays now gives a correct over-approximation.

$$\text{arrayblocks}(n) \quad \leq \quad \frac{n+5}{7} + (\log_8 n + 7) \tag{15}$$

We have a similar formula for arrays in OPENJDK, which uses continuous allocation with a small header by default (an array of n elements uses $4n + 8$ bytes, on a 32 bits target architecture). For each new JAVA VM a new specialisation for arrays needs to be added in order to correctly generate bounds for code using arrays.

¶These results are valid for data-types with a representation of four bytes. Alternate data-types (e.g. `byte`, `char`, `short`, `double`), can be calculated by multiplying the input n by a factor of $\frac{1}{4}$, $\frac{1}{2}$, $\frac{1}{2}$, 2 respectively.

3.4. *Simplification of bounds*

COSTA internally calculates the symbolic bounds without considering the format of the expression. The produced expressions are not necessary user friendly, e.g.:

$$\begin{aligned}
&nat(n)* \\
&\quad (nat(n) * (c(size(\texttt{java.lang.Object}, 1))+ \\
&\qquad\qquad c(size(\texttt{java.lang.Object}, 2)))+ \\
&\quad nat(n) * (c(size(\texttt{java.lang.Object}, 1))+ \\
&\qquad\qquad c(size(\texttt{java.lang.Object}, 2)))+ \\
&\quad nat(n) * (c(size(\texttt{java.lang.Object}, 1))+ \\
&\qquad\qquad c(size(\texttt{java.lang.Object}, 2))))
\end{aligned}$$

We implemented a recursive descent parser with reductions of mathematical expressions in order to make the expressions generated by COSTA more user readable. The result of an expression is not altered[||], but the formula is reordered and reduced to a more user friendly expression. The expression above is transformed into:

$$6n^2 * size(\texttt{java.lang.Object})$$

One can now easily see that the bound is quadratic. This simplification is built into RESANA and applied to all user-visible expressions.

3.5. *Example*

The complexity of calculating Fibonacci numbers is well known. The run-time complexity (in terms of methods calls) of calculating the nth Fibonacci value using a double recursion is related to the golden ratio $\varphi = \frac{1+\sqrt{5}}{2}$, which results in a complexity of $O(\varphi^n)$ method calls. Standard textbooks on complexity analysis use over-approximation, which results in a complexity of $O(2^n)$ for the same function. By adding an object allocation to each iteration, the heap consumption should be the same as the run-time complexity. The resulting code is given in Listing 10. Our tool annotates this function with the bound $(2^n - 1) * size(\texttt{java.lang.Object})$, matching the expected bound.

```
int fib(int n) {
    new Object();
    if (n < 2) return n;
    return fib(n-1) + fib(n-2);
}
```

Listing 10: Adaptation of the double recursive Fibonacci function, allocating an object in each call.

The nth Fibonacci number can also be calculated by using a single recursion, for which the complexity should be $O(n)$. The resulting code, with added object allocations, is listed in Listing 11. This single recursive function is annotated by our tool with a bound of $(n+1) * size(\texttt{java.lang.Object})$, also matching the expected complexity bound.

```
int fib_helper(int a, int b, int n) {
    new Object();
    if (n <= 0) return a;
    return fib_helper(b, a+b, n-1);
}
int fib(int n) {
    return fib_helper(0, 1, n);
}
```

Listing 11: Adaptation of the single recursive Fibonacci function, allocating an object in each call.

[||]Technically the output is altered a little bit as the allocation order, which only matters internally, is neglected. The allocation order is included in the `size` construct as the second argument. The nat function is also omitted for brevity, and should always be applied to variables.

4. STACK-SIZE ANALYSIS

The proposed method of stack analysis requires global knowledge of the program, including its data. A *data-flow-based static analyser* VERIFLUX is used to provide this knowledge [18] (see http://www.aicas.com/veriflux.html).

Analysis of *recursive methods* is a challenge in static evaluation of stack consumption. To deal with it, VERIFLUX's stack -size analysis relies on recursion-depth annotations. A recursion-depth annotation consists of an expression that evaluates to a natural number that is an upper bound on the number of nested recursive calls. Syntactically, recursion-depth annotations are provided as JML `measured_by` clauses. A `measured_by` expression is a usual symbolic expression like *a.length - 1*. VERIFLUX outputs the stack bound in bytes, which is the number computed from the annotations and the input data of the main method. If VERIFLUX discovers recursive methods that do not carry a recursion depth annotation, it uses a default recursion depth, which is a positive natural number or infinity. This number can be configured in the tool's GUI. In case the default recursion depth is configured to be infinity, the stack size analysis will report an infinite stack size for all threads that call recursive methods that do not carry a recursion-depth annotation.
Expressions for `measured_by` annotations are obtained using COSTA, which computes both:

- A symbolic upper bound on the depth of recursion (i.e. a "ranking function" for recursive calls) for a given method
- A symbolic upper bound on the number of calls of the method from itself.

The former corresponds to the height of the call tree, the latter represents the number of the nodes in the call tree. For instance, the depth of recursion for a typical implementation of the n-th Fibonacci number calculation belongs to $O(n)$, whereas the number of call belongs to $O(2^n)$. Both a ranking function and a bound on the number of recursive calls, can be used as `measured_by` expressions. The former and the latter coincide if the recursion branching factor $b < 2$. The number of calls leads to exponential over-approximation when $b \geq 2$.

Initially COSTA did not output ranking functions, even though they were a part of the tool its internal computations. The tool has been adjusted within the CHARTER project by adding an option that allows ranking functions to be shown.

Consider the method `fib`, computing the n-th Fibonacci number, in Listing 10. As expected, COSTA produces the ranking function $nat(n - 1)$. This represents the depth of the recursion tree. It is transformed by RESANA into the annotation `measured_by n-1`. The upper bound on the number of recursive calls that COSTA generates is $2 * (2^{nat(n-1)} - 1)$. This corresponds to the total number of nodes in the recursion tree.

A JAVA VM has two stacks: a JAVA stack and a native one. Interpreted code and dynamically generated code execute on the Java stack. External C libraries, JIT compiled (JAVA) code and Java functionality implemented natively execute on the native stack. Both have different stack usage characteristics. We consider Java stack usage while running the virtual machine in interpreted mode. While methods utilizing the native stack cannot be analysed automatically, the user can specify bounds in their JML contracts.

JAVA applications typically call methods from libraries. To obtain good stack-consumption bounds for such applications, one should provide stack-consumption bounds for library methods. In principle, library methods are analysed by CHARTER methodology in the same manner as applications, i.e. as the example above. However, analysis of libraries requires additional technical overhead, because of two issues: libraries are large and library methods call native routines.

4.1. Adjustments for analysis of libraries

Since a call to a library-method typically amounts to long chains of calls to other methods, the corresponding call graph becomes very large. The COSTA analysis is based on call graphs, so obtaining resource bounds in this case becomes unfeasible. Computations take too much time and/or at the end one obtains a huge unreadable symbolic expression. Therefore, it is advised to begin with analysis of the methods belonging to one strongly-connected component of the call graph. From our experience, COSTA performs it in reasonable time. Then continue with analysis of the methods who call already analysed ones, which annotations are used as *contracts*, et cetera. Eventually, all the library is analysed in a bottom-to-top manner.

Technically, *native stacks* are needed to cope with methods that are compiled to native machine code (for optimization purposes) and with native methods that are called through the Java Native Interface JNI (in order to access services provided by platform-specific native libraries). VERIFLUX does not address StackOverflowErrors due to overflows of native stacks. Since verification of C native methods is beyond of scope of this work, one has to rely on the known information about the behavior of these methods, i.e. corresponding contracts.

```
String toString(int i) {
   if (i == Integer.MIN_VALUE) return "-2147483648";
   int size = (i < 0) ? stringSize(−i) + 1 : stringSize(i);
   char[] buf = new char[size];
   getChars(i, size, buf);
   return MyString.valueOf(buf, 0, size);
}
```

Listing 12: The `toString` method from the `Integer` class in the JAVA standard library.

As an example for both issues, consider the `toString` method, which belongs to the `Integer` class and maps an integer number to a string, shown in Listing 12. Before running COSTA, place this method in the abstracted class `MyInteger`, that contains only `toString` and the methods called from it. Create the abstracted versions of the classes `StringIndexOutOfBoundsException` and `String`, that contain the methods called from `toString`, and the ones called from them, et cetera. COSTA produces a ranking function that symbolically depends on the costs of two native methods: `copyChars` and `cast2string`. If their contracts say that they do not call java methods (which is, indeed, the case for this example), their costs are turned into zeros by RESANA and the final `measured_by` expression is 0. This result can be approved by an accurate data-flow analysis of the method `toString` using pen and paper.

4.2. *Stack-size analysis by VeriFlux*

Consider the principles on which VERIFLUX its stack analysis is based. VERIFLUX computes an invocation graph, in which nodes correspond to methods and edges represent method invocations. Recursive method calls correspond to cycles in the graph. In order to eliminate cycles, one first computes the strongly connected components (SCCs) of the invocation graph[**]. Each SCC with more than zero nodes is then replaced by a single node that is annotated by *the sum of the sizes of all stack frames that correspond to nodes (i.e., method invocations) in that SCC, multiplied by the maximal recursion depth over all the nodes* (i.e., method invocations) in that SCC. The recursion depths are computed by evaluating the `measured_by` annotations of invoked methods or using the default recursion depth for methods that do not carry these annotations. All nodes that are not in an SCC with more than zero nodes are simply annotated by the size of the stack frame of the corresponding method invocation.

After merging each SCC, one is left with a directed acyclic graph (DAG), where each node is annotated with a positive integer. Let this annotation be called the stack-frame size of the node. To obtain the final result, VERIFLUX adds the stack frame size of the node to the maximum of the (recursively computed) stack sizes of its successor nodes. This can be achieved, for all nodes, in a depth-first traversal of the DAG.

From the user perspective, VERIFLUX performs stack analysis in the following way. The tool starts from the main method and evaluates the `measured_by` annotations of all called methods in an abstract environment. Variables (and expressions) in this environment are evaluated to intervals

[**]Recall that a strongly connected component of a directed graph is a sub-graph in which for any two nodes a and b, there is a path from a to b and vice versa.

that represent all possible values they may have according to data-flow analysis. For instance a variable n is replaced with the interval $[0, 21]$ if data-flow analysis shows that `fib`(n) will be called on n from 0 to 21.

The value that VERIFLUX outputs is an upper bound on the used stack in bytes, computed from the symbolic `measured_by` expressions and the input data of the main method. Note that VERIFLUX's computation of the abstract environment is approximate. In the worst case, VERIFLUX may have computed the abstract value 'Any' for some of the variables that occur in the `measured_by` expression. Then the concrete value of the `measured_by` expression evaluates to 'Any' as well. If a symbolic `measured_by` expression is not given, then a concrete default bound is involved, given by the user. The correctness of this given numerical upper bound is not checked, VERIFLUX simply uses this value in the analysis. The upper bounds computed by VERIFLUX are not tight, i.e., they may be higher than necessary.

Now, proceed with the Fibonacci example. Let it be called from the main method in Listing 13.

```
public static void main(String [] args) {
  fib(21);
}
```

Listing 13: Main method calling the `fib` method.

VERIFLUX computes the depth of recursion, which, as expected, is equal to 20. The upper bound on consumed stack space computed by VERIFLUX is 1156 bytes. This consists of 20 stack frames for the `fib` method, which use 56 bytes each, plus 36 bytes of stack space needed to call the method. Calling the same method with $n = 22$ results in a bound of 1212 bytes. This means that a stack overflow will not occur if 1156 and 1212 bytes of stack space are reserved for the main thread in the first and in the second case respectively.

To deal with virtual method invocations, VERIFLUX has an option "resolve opaque calls". When switched on, it considers all possible implementations or subclasses of a given interface or a superclass. If the analysis cannot resolve which virtual method is actually called, the maximum over the stack sizes of all those methods that are possibly called is used. Conceptually, the invocation graph will then have edges from the caller to all possibly called methods.

5. USER EXPERIENCE

We have combined all the CHARTER verification tools in a VIRTUALBOX image for easy installation. This image, the ECLIPSE plug-in and the source code, can be downloaded from the RESANA website[††].

The Dutch National Aerospace Laboratory NLR has used the VIRTUALBOX image in the development of a safety-critical avionics application. Their experience is described in [2]. They have selected the Environment Control System (ECS) on board an aircraft for evaluating the CHARTER tool-chain. The ECS is responsible for air conditioning and air pressurization. The application is written in REALTIME JAVA and runs on JAMAICAVM.

Before using the CHARTER tools, NLR did not determine any ranking functions for loops or memory-usage bounds, because manually devising them would require a very large effort. Now, thanks to RESANA, these bounds can be inferred relatively quickly, so the programmers now have a better understanding of the workings and hardware-requirements of their software. They applied RESANA for loop bound and heap space analysis. The tool was found to be easy to use. Their industrial user feedback has led to several (small but important) improvements of the RESANA tool. NLR has used the complete CHARTER tool-chain in their evaluation. The use of the tool set resulted in a considerable (89%) reduction of the errors that appeared during the testing phase of the project.

[††]http://resourceanalysis.cs.ru.nl/resana/

We ourselves have conducted several case studies as well. For loop bound analysis, three case studies were done. These are all parts of safety-critical Java systems, suggested as test cases by the CHARTER partners. The results are shown in Table I. As can be read from the table, we can handle roughly two-thirds of the loops found in the case studies. This means that we can infer a ranking function for these loops using our prototype and prove it using KEY. Almost all of the loops for which no ranking function could be inferred are loops for which the guard depends on a different program thread. An example of this is given in Listing 14. Analysing the temporal behaviour of such loops would require a fundamentally different analysis, which should take into account the code of all threads as well as the scheduling of threads. An example of a loop from the case studies for which a ranking function could be inferred and proved is given in Listing 15.

- **Collision detector case study.** The first case is the collision detector example from the paper "Provable Correct Loop bounds for Realtime Java Programs" by James Hunt et al [26]. This code stems from a safety-critical avionics application.
- **DIANA Package.** This package is developed in the FP6 project *Distributed, equipment Independent environment for Advanced avioNics Applications* (DIANA). The package is described in detail in [27].
- **CD_X Collision Detector package.** The CD_X Collision Detector package is a publicly available Real-Time Java Benchmark. It is described in [28].

	Nr. of loops	Analysable	Percentage
Hunt et al.	2	2	100%
DIANA	4	4	100%
CD_X	38	23	61%
Total	44	29	66%

Table I. Summary of the cases studied.

```
synchronized (lock) {
  while (producedFrames != consumedFrames) {
    try { lock.wait(); } catch (Exception e) {};
  }
}
```

Listing 14: Example loop from the immortal.FrameSynchronizer class in CD_X. No ranking function can be inferred for this loop, as it depends on another thread.

```
for (int i = 0; i < allocateArray.length; i++) {
  Object o = allocateArray[i];
  if (o != null) {
    currentRetention += ((byte[]) o).length;
  }
}
```

Listing 15: Example loop from the heap.MemoryAllocator class in CD_X. RESANA infers the ranking function $allocateArray.length - i$ for this loop, which can be proved using KEY.

During a course on software analysis, we have asked undergraduate students to perform a series of exercises using RESANA. Students successfully used the tool to infer ranking functions, heap bounds and stack bounds for various examples. Also, they performed a small case study on the code of Pygmy (a small web-server). Again, ranking functions could be generated for roughly two thirds of the loops. Similar exercises were also given to PhD students at the 2013 IPA[‡‡] school on Software Engineering and Technology, who found the tool to be very useful.

[‡‡]Institute for Programming research and Algorithmics: http://www.win.tue.nl/ipa/

6. RELATED WORK

This article is an extended version of an earlier paper [3]. It was extended on several distinct points. The extensions to the loop-bound inference method discussed in Section 2.4 (rational or real coefficients), Section 2.5 (branch-splitting) and Section 2.6 (piecewise ranking functions) were already described in more primitive form in [14]. Condition jumping and the corresponding extension to the loop-bound inference method, described in Section 2.7, were not previously published. There are also some new contributions with respect to heap analysis. First, verification of ranking functions derived with our experimental method using QEPCAD was implemented within COSTA. This is described in Section 3.1. Furthermore, a specialization of the results for OPENJDK was made (previously, only a specialization for JAMAICAVM was available), see Section 3.3. All these extensions are implemented in the current version of RESANA.

The polynomial interpolation based technique was successfully applied in the analysis of output-on-input data-structure size relations for functions in a functional language, in [5], [29], [30], [31], [32] and [33]. This method can for instance be used to determine that if the append function gets two lists of lengths n and m as input, it will return a list of length $n + m$.

6.1. *Loop-Bound Analysis*

Hunt et al. discuss the expression of manually conceived ranking functions in JML, their verification using KEY and the combination with data-flow analysis in [26]. What is "missing" in the method is the automated inference of ranking functions, which RESANA supplies.

In [34], an approach that is similar to ours is taken, in the combination of COSTA with the KEY tool. The results that COSTA gives are output as JML annotations, that may then be verified using KEY.

Various other research results on bounding the number of loop iterations are described in the literature. However, most approaches generate concrete (numerical) bounds [35, 36, 37], as opposed to *symbolic* bounds. The methods that are able to infer symbolic loop bounds are limited to either bounds that depend linearly on program variables (the procedure described in this paper infers polynomial bounds) [25] or that are constructed from monotonic subformulae [38, 6].

Several syntactical methods are discussed [39, 40], that will be more efficient for simple cases, but less general. Our procedure can be seen as complementary to those methods. In case a syntactical method is not applicable to a certain loop, our more general method can be used.

To generate algebraic loop invariants, Sharma et al. [41] use a procedure which, as our loop-bound inference algorithm, employs interpolation and separated inference and verification phases. They refer to their algorithm as *guess-and-check*, as it employs a non-sound inference phase and a verification phase. In the inference phase, the program is executed on data from unit tests and results are interpolated. For checking the invariants they use an SMT solver. The main difference to our work is that they search for so-called algebraic *invariants*, which are defined as algebraic *equalities* over program variables, whereas we search for a specific *variant* (a ranking function) specifying the number of remaining iterations of the loop, the value of which is required to decrease on each iteration. This ranking function implies an algebraic *inequality* as invariant.

6.2. *Time Performance Analysis*

There a a number of parallels of our work with time performance analysis. This can be average execution time analysis or , more common, Worst Case Execution Time (*WCET*) analysis. As already mentioned, loop-bound inference can be used for time analysis, in particular for WCET analysis. Depending on the cost function associated with each iteration of the loop, one can compute a memory bound or a timing bound. To properly use this for WCET analysis one has to incorporate extra analysis of e.g. cache behaviour, context switches, etc. to precisely approximate the worst case execution time as is done in [42, 43].

As memory allocators and cache policies are rather slow and unpredictable, the number and the amount of memory allocated have an impact on performance [43]. One has to resort to special means to alleviate these problems [44]. Our heap analysis can also be used to gain insight into the allocations of a program. This can help with reducing the number and amount of allocations in a program, which can lead to smaller worst case execution times.

6.3. Heap-Space Usage Analysis

We have taken the COSTA system [15] as our point of reference. The authors have recently improved [45] the precision of PUBS, its recurrence solver, by considering upper and lower bounds to the cost of each loop iteration. In a different direction, COSTA has improved its memory analysis in order to take different models of garbage collection into account [23]. However, the authors claim that this extension does not require any changes to the recurrence solver PUBS. Thus, the techniques presented in Section 3.1 should fit with these extensions.

In the field of functional languages, a seminal paper on static inference of memory bounds is [46]. A special type inference algorithm generates a set of linear constraints which, if satisfiable, specify a safe linear bound on the heap consumption. One of the authors extended this type system in [47, 7] in order to infer multivariate polynomial bounds. Surprisingly, the constraints resulting from the new type system are still linear.

6.4. Stack-Size Analysis

In practice, stack usage in JAVA is often measured by instrumenting or transforming the source code so that it counts consumed resources (and computes other relevant information) on the inputs of the original code. To our knowledge, there are two commercial tools that perform JAVA stack analysis: *Coverity Static Analyzer* and *Klockwork*, with its *kwstackoverflow*. Another tool, *GNATStack*, analyses object-oriented applications, automatically determining maximum stack usage on code that uses dynamic dispatching in Ada and C++.

In [48], a static stack-bound analysis for abstract JAVA bytecode is described. The described method considers JAVA bytecode with recovered high-level control structures (conditionals and while-loops). The inference process is divided into three key stages: frame-bound inference, abstract-state inference and stack-bound inference. Recall that a frame is a piece of stack reserved for each method invocation. Each stage applies a corresponding set of inference rules. In these rules the authors use *Presburger (linear) arithmetic formulae* to describe states of programs. It is stated that an implementation is under development.

7. CONCLUSION AND FUTURE WORK

To assist in making resource analysis practical, we have introduced new techniques and combined these techniques in our new tool, RESANA. Complex loop, heap and stack bounds can be inferred in an integrated way within the ECLIPSE IDE. Bounds can be inferred that are specific for the underlying virtual machine (shown both for JAMAICAVM and OPENJDK).

Obviously, a full resource analysis tool would also need to build in an elaborate time analysis. For now, we will rely on other tools to provide such information. The ability to infer resource bounds contributes to improving the development process of producing real-time safety-critical systems both with respect to ease of development and with respect to improved reliability. The Dutch National Aerospace Laboratory (NLR) has successfully used RESANA in the development of a demonstrator safety-critical REALTIME JAVA avionics application.

Future Work The capabilities of time analysis tools could of course be incorporated in our tool or it could be made easy to switch from our tool to time analysis tools and to exchange information. Another direction of future research could be to include work on other kinds of resources that are consumed, e.g. also inferring and proving energy related properties of JAVA programs might be important. Furthermore, one could define, instead of a single overall memory bound for the complete run-time of a program, a time dependent memory bound which gives a bound for the consumption on a certain moment in the execution of a program. Such a time dependent bound is called a *live* memory bound. Together with information on synchronisation moments, this opens up the possibility to derive more precise memory bounds by adding upper bounds of processes in the periods between synchronisation moments.

ACKNOWLEDGEMENTS

We would like to thank James Hunt, Isabel Tonin and Christian Haack of AICAS for their help and insights in developing the stack-size analysis. Also, we would like to thank Gosse Wedzinga and Klaas Wiegmink at NLR for their useful feedback on RESANA use in practice. We thank Sebastiaan Joosten for his valuable advice on elements of this paper. We would like to thank Samir Genaim for his technical support regarding COSTA and we thank the anonymous reviewers for their important suggestions in improving this paper.

REFERENCES

1. de Mol MJ, Rensink A, Hunt JJ. Graph transforming java data. *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012), Talinn, Estonia*, *Lecture Notes in Computer Science*, vol. 7212, Springer Verlag: London, 2012; 209–223.
2. Wedzinga G, Wiegmink K. Using charter tools to develop a safety-critical avionics application in java. *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, ACM: New York, NY, USA, 2012; 125–134, doi:10.1145/2388936.2388958. URL http://doi.acm.org/10.1145/2388936.2388958.
3. Kersten R, Shkaravska O, van Gastel B, Montenegro M, van Eekelen M. Making resource analysis practical for Real-Time Java. *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, ACM: New York, NY, USA, 2012; 135–144, doi:10.1145/2388936.2388959. URL http://doi.acm.org/10.1145/2388936.2388959.
4. Albert E, Arenas P, Genaim S, Puebla G. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning* February 2011; **46**(2):161–203.
5. van Kesteren R, Shkaravska O, van Eekelen M. Inferring static non-monotonically sized types through testing. *16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07), Paris, France*, *Electronic Notes in Theoretical Computer Science*, vol. 216C, 2008; 45–63.
6. Gulwani S, Zuleger F. The reachability-bound problem. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, ACM: New York, NY, USA, 2010; 292–304.
7. Hoffmann J, Aehlig K, Hofmann M. Multivariate amortized resource analysis. *POPL'11*, Ball T, Sagiv M (eds.), ACM, 2011; 357–370.
8. Amadio RM. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae* August 2005; **65**(1-2):29–60. URL http://portal.acm.org/citation.cfm?id=1227143.1227146.
9. de Dios J, Montenegro M, Peña R. Certified absence of dangling pointers in a language with explicit deallocation. *8th International Conference on Integrated Formal Methods, IFM 2010*, LNCS 6396, Springer, 2010; 305–319.
10. van Eekelen M, Shkaravska O, van Kesteren R, Jacobs B, Poll E, Smetsers S. AHA: Amortized Heap Space Usage Analysis. *Selected Papers of the 8^{th} International Symposium on Trends in Functional Programming (TFP'07), New York, USA*, Morazán M (ed.), Intellect Publishers, UK, 2007; 36–53.
11. Leavens GT, Poll E, Clifton C, Cheon Y, Ruby C, Cok D, Müller P, Kiniry J, Chalin P. *JML Reference Manual. Draft Revision 1.200* Feb 2007.
12. Beckert B, Hähnle R, Schmitt PH ((eds.)). *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334, Springer, 2007.
13. Brown CW. QEPCAD B: a program for computing with semi-algebraic sets using CADs. *SIGSAM Bull.* Dec 2003; **37**(4):97–108, doi:10.1145/968708.968710. URL http://doi.acm.org/10.1145/968708.968710.
14. Shkaravska O, Kersten R, Van Eekelen M. Test-based inference of polynomial loop-bound functions. *PPPJ'10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, Krall A, Mössenböck H (eds.), ACM Digital Proceedings Series, 2010; 99–108.
15. Albert E, Arenas P, Genaim S, Puebla G, Zanardini D. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. *Formal Methods for Components and Objects*, *Lecture Notes in Computer Science*, vol. 5382, de Boer F, Bonsangue M, Graf S, de Roever W (eds.). Springer, 2008; 113–132.
16. Siebert F. Hard realtime garbage collection in modern object oriented programming languages. PhD Thesis, University of Karlsruhe 2002.
17. Montenegro M, Shkaravska O, van Eekelen M, Peña R. Interpolation-based height analysis for improving a recurrence solver. *Proceedings of the 2nd Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2011*, LNCS 7177, Springer, 2012; 36–53.
18. Hunt JJ, Tonin I, Siebert FB. Using global data flow analysis on bytecode to aid worst case execution time analysis for real-time java programs. *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, ACM: New York, NY, USA, 2008; 97–105, doi:http://doi.acm.org/10.1145/1434790.1434806.
19. Chui CK, Lai MJ. Vandermonde determinants and lagrange interpolation in R^s. *Nonlinear and convex analysis* 1987; :23–35.
20. Poll E, Chalin P, Cok D, Kiniry J, Leavens GT. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. *In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, volume 4111 of LNCS*, Springer, 2006; 342–363.

21. King JC. Symbolic execution and program testing. *Commun. ACM* July 1976; **19**:385–394.
22. Ranise S, Tinelli C. The smt-lib format: An initial proposal 2003; URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.123.9580`.
23. Albert E, Genaim S, Gómez-Zamalloa M. Parametric inference of memory requirements for garbage collected languages. *ISMM'10*, Vitek J, Lea D (eds.), ACM, 2010; 121–130.
24. Wegbreit B. Mechanical program analysis. *Commun. ACM* 1975; **18**(9):528–539.
25. Podelski A, Rybalchenko A. A complete method for the synthesis of linear ranking functions. *Verification, Model Checking, and Abstract Interpretation*, *Lecture Notes in Computer Science*, vol. 2937, Steffen B, Levi G (eds.). Springer Berlin / Heidelberg, 2004; 465–486.
26. Hunt JJ, Siebert FB, Schmitt PH, Tonin I. Provably correct loops bounds for realtime java programs. *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, ACM: New York, NY, USA, 2006; 162–169, doi:http://doi.acm.org/10.1145/1167999.1168026.
27. Schoofs T, Jenn E, Leriche S, Nilsen K, Gauthier L, Richard-Foy M. Use of PERC Pico in the AIDA avionics platform. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM, 2009; 169–178.
28. Kalibera T, Hagelberg J, Pizlo F, Plsek A, Titzer B, Vitek J. Cd$_x$: a family of real-time java benchmarks. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009, Madrid, Spain, September 23-25, 2009*, ACM, 2009; 41–50.
29. Shkaravska O, van Kesteren R, van Eekelen M. Polynomial Size Analysis for First-Order Functions. *Typed Lambda Calculi and Applications (TLCA'2007), Paris, France*, *LNCS*, vol. 4583, Rocca SRD (ed.), Springer, 2007; 351–366.
30. Shkaravska O, van Eekelen M, Tamalet A. Collected size semantics for functional programs over lists. *Proceedings of the 20th international conference on Implementation and application of functional languages*, IFL'08, Springer-Verlag: Berlin, Heidelberg, 2011; 118–137. URL `http://dl.acm.org/citation.cfm?id=2044476.2044483`.
31. Shkaravska O, van Eekelen M, van Kesteren R. Polynomial size analysis of first-order shapely functions. *Logic in Computer Science* 2009; **2:10**(5). URL `http://arxiv.org/abs/arxiv:0902.2073`.
32. Tamalet A, Shkaravska O, van Eekelen M. Size analysis of algebraic data types. *Trends in Functional Programming*, *Trends in Functional Programming*, vol. 9, Achten P, Koopman P, Morazán M (eds.), Intellect, 2009; 33–48. ISBN 978-1-84150-277-9.
33. Gobi A, Shkaravska O, van Eekelen M. Higher-order size checking without subtyping. *Proceedings of the 13th International Symposium on Trends in functional Programming (TFP2012)*, *Lecture Notes in Computer Science*, vol. 7829, Loidl HW, Hammond K (eds.), Springer, 2013; 53–68.
34. Albert E, Bubel R, Genaim S, Hähnle R, Puebla G, Román-Díez G. Verified resource guarantees using COSTA and KeY. *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, ACM: New York, NY, USA, 2011; 73–76.
35. Ermedahl A, Sandberg C, Gustafsson J, Bygde S, Lisper B. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Rochange C (ed.), Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany: Dagstuhl, Germany, 2007.
36. Lokuciejewski P, Cordes D, Falk H, Marwedel P. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE Computer Society: Washington, DC, USA, 2009; 136–146.
37. De Michiel M, Bonenfant A, Cassé H, Sainrat P. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. *RTCSA '08: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, IEEE Computer Society: Washington, DC, USA, 2008; 161–166.
38. Gulwani S. SPEED: Symbolic complexity bound analysis. *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, Springer-Verlag: Berlin, Heidelberg, 2009; 51–62.
39. Fulara J, Jakubczyk K. Practically applicable formal methods. *SOFSEM '10: Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, Springer, 2010; 407–418.
40. Gulwani S, Jain S, Koskinen E. Control-flow refinement and progress invariants for bound analysis. *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ACM: New York, NY, USA, 2009; 375–385.
41. Sharma R, Gupta S, Hariharan B, Aiken A, Liang P, Nori A. A data driven approach for algebraic loop invariants. *Programming Languages and Systems*, *Lecture Notes in Computer Science*, vol. 7792, Felleisen M, Gardner P (eds.). Springer Berlin Heidelberg, 2013; 574–592, doi:10.1007/978-3-642-37036-6_31. URL `http://dx.doi.org/10.1007/978-3-642-37036-6_31`.
42. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, *et al.*. The worst-case execution-time problemoverview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* May 2008; **7**(3):36:1–36:53, doi:10.1145/1347375.1347389. URL `http://doi.acm.org/10.1145/1347375.1347389`.
43. Reineke J, Grund D, Berg C, Wilhelm R. Timing predictability of cache replacement policies. *Real-Time Systems* November 2007; **37**(2):99–122, doi:10.1007/s11241-007-9032-3. URL `http://rw4.cs.uni-saarland.de/~grund/papers/rts07-predictability.pdf`.
44. Herter J, Backes P, Haupenthal F, Reineke J. CAMA: A predictable cache-aware memory allocator. *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS '11)*, IEEE Computer Society, 2011. URL `http://rw4.cs.uni-sb.de/~jherter/papers/CAMAecrts11.pdf`.
45. Albert E, Genaim S, Masud AN. More precise yet widely applicable cost analysis. *VMCAI'11*, *Lecture Notes in Computer Science*, vol. 6538, Jhala R, Schmidt DA (eds.), Springer, 2011; 38–53.

46. Hofmann M, Jost S. Static prediction of heap space usage for first-order functional programs. *POPL'03*, ACM Press, 2003; 185–197, doi:http://doi.acm.org/10.1145/604131.604148.
47. Hoffmann J, Hofmann M. Amortized Resource Analysis with Polynomial Potential. A Static Inference of Polynomial Bounds for Functional Programs. *ESOP'10, LNCS 6012*, Springer, 2010; 287–306.
48. Wang S, Qiu Z, Qin S, Chin WN. Stack bound inference for abstract java bytecode. *Proceedings of the 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE '10, 2010; 57–66, doi: 10.1109/TASE.2010.24. URL http://dx.doi.org/10.1109/TASE.2010.24.