Making Resource Analysis Practical for Real-Time Java

Rody Kersten
Institute for Computing and Information Sciences
Radboud University Nijmegen
r.kersten@cs.ru.nl

Olha Shkaravska
Institute for Computing and Information Sciences
Radboud University Nijmegen
shkarav@cs.ru.nl

Bernard van Gastel
School of Computer Science
Open University
bernard.vangastel@ou.nl

Manuel Montenegro
Departamento de Sistemas Informáticos y Computación
Universidad Complutense
Madrid
montenegro@fdi.ucm.es

Marko van Eekelen
Institute for Computing and Information Sciences,
Radboud University Nijmegen
and School of Computer Science, Open University
marko@cs.ru.nl

ABSTRACT

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 these resources during the development process of the code can be of great value.

Recent research results have advanced the state of the art of resource consumption analysis. In this paper we present a tool that makes it possible to apply these research results in practice for real-time systems enabling JAVA developers to analyse loop bounds, bounds on heap size and 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 tool RESANA that is the result of this effort. The tool 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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, Reliability, Algorithms

Keywords

Resource analysis, Polynomial interpolation, Ranking function, Heap bounds, Stack bounds

JTRES 2012 October 24-26, 2012, Copenhagen, Denmark

1. INTRODUCTION

Both in industry and in academia there is an increasing interest in more detailed resource analysis bounds than orders of complexity. Focus in correctness for industrial critical systems 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. A variety of different techniques have been developed independently not only on the language level but also on the byte code level [1]. Researchers use polynomial interpolation [31], reachability-bound analysis [16], amortization [17], polynomial quasi-interpretation [6] and new language features such as programmer-controlled destruction and copying of data structures [10]. Of course, such analyses are undecidable in general. In practice, however, an increasingly large set of problems can be handled.

This research has been performed in the context of the EU Artemis CHARTER (**C**ritical and **H**igh **A**ssurance **R**equirements **T**ransformed through **E**ngineering **R**igour) project. For safety-critical areas such as avionics, surveillance, automotive and health-care, the project targets REALTIME JAVA in the context of a model driven development approach. Being able to guarantee bounds for resource consumption is a vital aspect for safety-critical systems. In this paper, we focus on the JAVA language and on resource consumption properties related to time, 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 thorough analysis is needed.

With the goals of making these results applicable in practice, our 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[1], that contains a general process which has two phases.

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 [22].

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 [7] or the QEPCAD algebraic decomposition tool [8]. 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 tool ResAna allows three different kinds of analysis.

Loop Bound Analysis An expression that gives an upper bound for the number times a loop is executed may be derived and verified using the integrated combination [27] of the tools ResAna and KeY.

Heap Bound Analysis An expression for an upper bound of the consumed heap is derived using ResAna extended with a variant of the external tool COSTA [2]. The tool COSTA has been adapted to produce accurate values for the real-time JamaicaVM virtual machine [30]. Furthermore, the capabilities of the tool COSTA have been enlarged through the internal use of interpolation technology [24].

Stack Bound Analysis An expression for an 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 [21] 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 safety of critical real-time software applications.

The *contribution of this paper* is threefold. First of all, it contains the first integral description of the tool ResAna that makes resource analysis practical by comprising many features assisting real-time developers in producing reliable safety critical code that stays within the required loop, heap and stack bounds. Secondly, it contains the first explanation of a technique that is used to achieve virtual-machine-dedicated, tight heap bounds for real-time programs dealing with arrays. Thirdly, it describes how the information of the tools COSTA, QEPCAD/KeY and VeriFlux can be combined in ResAna to achieve sound stack bounds.

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

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

2. LOOP-BOUND ANALYSIS

In order to prove termination of a piece of software or, even harder, 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 user 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.

```
1 while (i < 15)
2   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. This procedure was first presented by Shkaravska et al in [27]. 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 will introduce polynomial interpolation as used for ranking function inference in Sect. 2.1. Section 2.2 describes the inference method. In Sect. 2.3, a quadratic example is given. Finally, the soundness of the method is discussed in Sect. 2.4. Another application of our polynomial interpolation method is discussed in Sect. 3.1.

2.1 Polynomial Interpolation

When the result of a polynomial function is known for certain test values, the values of its coefficients can be derived. Such a polynomial, which *interpolates* the test results, exists and is unique under some conditions on the data, which are explored in polynomial-interpolation theory [9].

For 1-variable interpolation this condition is well-known: all the test-nodes must be different. For *multivariate* data, more care must be taken in selecting the right test-nodes.

In [9], three node configurations are given that ensure the existence of a unique multivariate polynomial interpolation. We use Node Configuration A (**NCA**) for ranking function inference. Two-dimensional **NCA** can be defined as follows. Remember that a polynomial $p(z_1, \ldots, z_k)$ of degree d and dimension k (the number of variables) has $N_d^k = \binom{d+k}{k}$ coefficients. This is the number of test-nodes that we need.
*N_d^2 nodes forming a set $W \subset \mathcal{R}^2$ lie in a 2-dimensional **NCA** if there exist lines $\gamma_1, \ldots, \gamma_{d+1}$ in the space $\mathcal{R}^2$, such that $d+1$ nodes of W lie on γ_{d+1} and d nodes of W lie on $\gamma_d \setminus \gamma_{d+1}$, ..., and finally 1 node of W lies on $\gamma_1 \setminus (\gamma_2 \cup \ldots \cup \gamma_{d+1})$.*

For dimensions $k > 2$, **NCA** is defined inductively on k. A set of N_d^k nodes is in **NCA** in $\mathcal{R}^k$ if and only if

- *there is a $(k-1)$-dimensional hyperplane such that it contains N_d^{k-1} of the given nodes lying in $(k-1)$-dimensional **NCA** for the degree d,*
- *for any $0 \leq i \leq d$, there is a $(k-1)$-dimensional hyperplane such that it contains N_{d-i}^{k-1} nodes, lying in $(k-1)$-dimensional **NCA** for the degree $d-i$ and these nodes do not lie on the previous hyperplanes,*
- *thus, the remaining 1 node lies on the remaining hyperplane and does not belong to the previous ones.*

A typical instance of **NCA** is a k-dimensional *grid*.

2.2 Test-Based Inference of Polynomial Ranking Functions for Loops

In [27], 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:

$$\bigwedge_{i=1}^{n_i} (e_{li}\ \mathbf{b}\ e_{ri})$$

with $\mathbf{b} \in \{<, >, =, \neq, \leq, \geq\}$.

The method works 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. To ensure the existence of a unique interpolation, the test nodes are chosen to lie in Node Configuration A (**NCA**). Also, test-nodes must satisfy the guard of the considered loop. An algorithm for node search is presented in [27].

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 explication of the tree height by the programmer.

2.3 Quadratic Example

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

```
1 while (a > 0 && c <= b && c > 0) {
2   if (c == b) { a--; c = 0; }
3   c++;
4 }
```

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

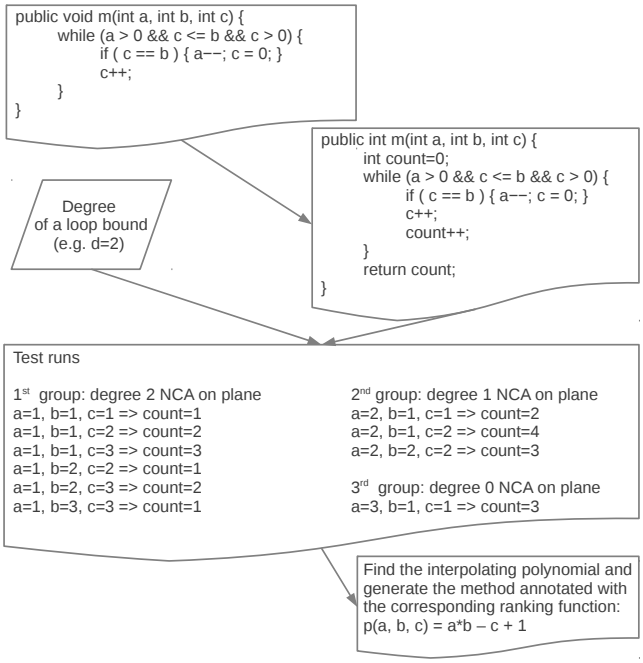

Figure 1: Test-based inference method applied to the example from Listing 2.

The inference of a ranking function for the loop in Listing 2 is depicted in Fig. 1. First, the loop is instrumented with a counter. The user inputs the expected degree 2 of the polynomial ranking function. Since there are 3 variables, a set of $N_2^3 = 10$ 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.

2.4 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 [26]. 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 [22]. 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 ex-

presses 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 [7]. The procedure described here should be used in conjunction with such a prover to provide soundness.

```
1 /*@
2   decreases (a > 0) && (c <= b) && (c > 0) ?
3     a * b - c + 1 : 0;
4 */
5 while (a > 0 && c <= b && c > 0) {
6   if (c == b) { a--; c = 0; }
7   c++;
8 }
```

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

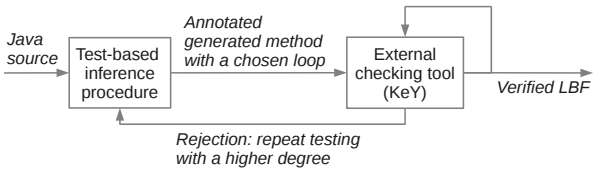

Figure 2: The basic inference procedure from a bird's eye view: infer-and-check cycle.

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.

3. HEAP-SPACE USAGE ANALYSIS

ResAna's heap consumption analysis is based on the tool COSTA [2], 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 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: a user can select 'scope', 'reachability', 'liveness', or none at all [4]. This functionality is retained in ResAna. Inside Java Realtime Threads no garbage collection is applied, so a user can select to ignore garbage collection. In normal Java one has to match the type of garbage collector used in the target virtual machine to a garbage collection model of COSTA in order to generate a sound bound.

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, due to Wegbreit [34], 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 4.

```
1 while (x <= y) {
2   new Object();
3   if (...) x = x + 1; else y = y - 2;
4 }
```

Listing 4: 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:

$$\begin{array}{ll} T(x,y) = 0 & \{x \geq y+1\} \\ T(x,y) = c + T(x',y) & \{x \leq y, x' = x+1\} \\ T(x,y) = c + T(x,y') & \{x \leq y, y' = y-2\} \end{array}$$

where c denotes the constant $c(size(\texttt{java.lang.Object}))$, i.e. the memory cost of creating an instance of `Object`. The COSTA system provides the recurrence solver PUBS [1], which computes the following closed-form:

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

This is an upper-bound to the values of $T(x, y)$ given above. The notation $nat(expr)$ abbreviates $max\{0, expr\}$. 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 $y - x + 1$:

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

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:

$$\begin{array}{ll} R(x,y) = c & \{x = 0, y = 0\} \\ R(x,y) = c + R(x',y') & \{x > 0, x' = x - 1, y' = x - 1\} \\ R(x,y) = c + R(x,y') & \{x \geq 0, y > 0, y' = y - 1\} \end{array}$$

This 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. 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 2.2. The main difficulty lies in non-determinism. When evaluating a CRS with a set of input values, the number of unfolding sequences may be infinite, even if the evaluation yields a finite number of results. We have addressed this problem by determining the sets of inputs in which the maximum number of unfoldings of the CRS are known. For instance, in the CRS above defining T, we define A_i (for each $i \in \mathcal{N}$) as the set of pairs (x,y) such that the evaluation of $T(x,y)$ does not require more than i unfoldings. We have found a characterization of these sets in terms of the guards of the CRS (see [24] for more details). Each set can be described as a disjoint union of convex polyhedra, and this description can be computed by using quantifier elimination techniques. For instance, in our CRS above describing T we get the following characterizations:

$$A_i = \{(x,y) \in \mathcal{N}^2 \mid x \geq y + 1 - i\} \text{ for each } i \in \mathcal{N}$$

We use a gradient-based approach for selecting the interpolating nodes from the A_i sets. The algorithm involves the search of *climbing paths* starting at the A_0 set, and minimizing the distance between A_i and $A_{i+1} \backslash A_i$ for each $i \in \mathcal{N}$.

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 predicates 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 statements for checking that $y - x + 1$ is an upper bound to $T(x,y)$:

$$\begin{array}{c} \forall x, x', y : x \leq y \wedge x' = x + 1 \Rightarrow y - x + 1 \geq 1 + y - x' + 1 \\ \forall x, y, y' : x \leq y \wedge y' = y - 2 \Rightarrow y - x + 1 \geq 1 + y' - x + 1 \\ \forall x, y : y - x + 1 \leq 0 \Rightarrow x \geq y + 1 \end{array}$$

If these generated predicates hold, then $y - x + 1$ is indeed an upper bound to $T(x,y)$. In order to check such inequalities the QEPCAD tool [8] can be used. QEPCAD proves all three logical statements above. The integration of COSTA and QEPCAD is subject of future work.

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-bits 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 [30]. An example array structure is shown in Figure 3. 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.

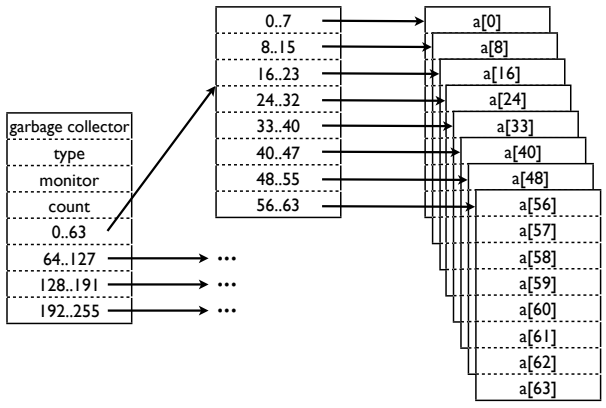

Figure 3: Graphical representation of a JAMAICAVM array of size n, with $33 \leq n \leq 255$, with $a[i]$ representing the contents of the array. Allocating an array of 63 elements takes 10 blocks.

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. The exact sizes are retrieved from the target VM by means of a specially generated program. This generated program depends on the Scoped Memory extensions of RealtimeJava. For each new Java VM a new specialisation needs to be added.

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 1, 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[2], which correspond to the size of pointers used in the tree structure pointing to the leafs, resulting in a cleaner formula.

$$\begin{aligned} \text{arrayblocks}(1..8) &= 1 \\ \text{arrayblocks}(n) &= \left\lceil \frac{n}{8} \right\rceil + \text{arrayblocks}(\left\lceil \frac{n}{8} \right\rceil) \end{aligned} \quad (1)$$

By transforming the formula to an over-approximation (by replacing $\lceil \frac{n}{8} \rceil$ with $\frac{n+7}{8}$), we can solve it as a recurrence equation, which after adjustments for the start cases results in Equation 2. 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.

$$\text{arrayblocks}(n) \leq \frac{n+5}{7} + (\log_8 n + 7) \quad (2)$$

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.:

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

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[3], 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.

[2]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]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.

3.5 Example

The complexity of calculating Fibonacci numbers is well known. The runtime complexity (in terms of methods calls) of calculating the nth Fibonnaci 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 runtime complexity. The resulting code is given in Listing 5. Our tool annotates this function with the bound $(2^n - 1) * size(\texttt{java.lang.Object})$, matching the expected bound.

```
1 int fib(int n) {
2     new Object();
3     if (n < 2)
4         return n;
5     return fib(n-1) + fib(n-2);
6 }
```

Listing 5: Adaptation of the double recursive Fibonacci function, allocating an object in each call.

The nth Fibonnaci 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 6. 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.

```
1 int fib_helper(int a, int b, int n) {
2     new Object();
3     if (n <= 0)
4         return a;
5     return fib_helper(b, a+b, n-1);
6 }
7 int fib(int n) {
8     return fib_helper(0, 1, n);
9 }
```

Listing 6: Adaptation of the single recursive Fibonacci function, allocating an object in each call.

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 [21] (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 5. 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.

Note that `measured_by` expressions obtained by polynomial interpolation (via switching-on the corresponding option in COSTA) are unsound in general. Automatic verification of such expressions can be done via QEPCAD or KeY. Technical integration of these tools into the stack-analysis procedure of ResAna is work in progress.

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.

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 7.

```
1 String toString(int i) {
2    if (i == Integer.MIN\_VALUE)
3        return "-2147483648";
4    int size = (i < 0) ?
5        stringSize(-i) + 1 : stringSize(i);
6    char[] buf = new char[size];
7    getChars(i, size, buf);
8    return MyString.valueOf(buf, 0, size);
9 }
```

Listing 7: The `toString` method from the `Integer` class in the Java standard library.

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[4]. 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*

[4]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.

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 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 8.

```
1 public static void main(String [] args) {
2   fib(21);
3 }
```

Listing 8: 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[5].

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 [33]. The application is written in Realtime Java and runs on JamaicaVM. The ResAna tool was found to be easy to use. Their user feedback has led to several improvements of the ResAna tool. In their avionics demonstrator project (an environment control system) they used the CHARTER tool set. They applied ResAna for loop bound and heap space analysis.

6. RELATED WORK

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 [29],[31] and [28]. 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 [20]. What is "missing" in the method is the automated inference of ranking functions, which ResAna supplies.

In [3], 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 [12, 23, 11], 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 [14, 16].

Several syntactical methods are discussed [13, 15], 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.

6.2 Heap-Space Usage Analysis

We have taken the COSTA system [2] as our point of reference. The authors have recently improved [5] 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 [4]. 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.

[5] `http://resourceanalysis.cs.ru.nl/resana/`

In the field of functional languages, a seminal paper on static inference of memory bounds is [19]. 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 [18, 17] in order to infer multivariate polynomial bounds. Surprisingly, the constraints resulting from the new type system are still linear.

6.3 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* (see `http://www.coverity.com`) and *Klockwork*, with its *kwstackoverflow* (see `http://www.klocwork.com/`). Another tool, *GNATStack*, analyses object-oriented applications, automatically determining maximum stack usage on code that uses dynamic dispatching in Ada and C++ (see `http://www.adacore.com/gnatstack/`).

In [32], 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

To make 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.

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.

8. 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 advise on elements of this paper. Finally, we would like to thank Samir Genaim for his technical support regarding COSTA.

This research is partly funded by the CHARTER project grant: ARTEMIS-2008-1-100039.

9. REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In F. de Boer, M. Bonsangue, S. Graf, and W. de Roever, editors, *Formal Methods for Components and Objects*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2008.

[3] E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. Verified resource guarantees using COSTA and KeY. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 73–76, New York, NY, USA, 2011. ACM.

[4] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In J. Vitek and D. Lea, editors, *ISMM'10*, pages 121–130. ACM, 2010.

[5] E. Albert, S. Genaim, and A. N. Masud. More precise yet widely applicable cost analysis. In R. Jhala and D. A. Schmidt, editors, *VMCAI'11*, volume 6538 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2011.

[6] R. M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, August 2005.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

[8] C. W. Brown. QEPCAD B: a program for computing with semi-algebraic sets using CADs. *SIGSAM Bull.*, 37(4):97–108, Dec. 2003.

[9] C. K. Chui and M.-J. Lai. Vandermonde determinants and lagrange interpolation in R^s. *Nonlinear and convex analysis*, pages 23–35, 1987.

[10] J. de Dios, M. Montenegro, and R. Peña. Certified absence of dangling pointers in a language with explicit deallocation. In *8th International Conference on Integrated Formal Methods, IFM 2010*, LNCS 6396, pages 305–319. Springer, 2010.

[11] M. De Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *RTCSA '08: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 161–166, Washington, DC, USA, 2008. IEEE Computer Society.

[12] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In C. Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[13] J. Fulara and K. Jakubczyk. Practically applicable formal methods. In *SOFSEM '10: Proceedings of the 36th Conference on Current Trends in Theory and*

Practice of Computer Science, pages 407–418. Springer, 2010.

[14] S. Gulwani. SPEED: Symbolic complexity bound analysis. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 51–62, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 375–385, New York, NY, USA, 2009. ACM.

[16] S. Gulwani and F. Zuleger. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 292–304, New York, NY, USA, 2010. ACM.

[17] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In T. Ball and M. Sagiv, editors, *POPL'11*, pages 357–370. ACM, 2011.

[18] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. A Static Inference of Polynomial Bounds for Functional Programs. In *ESOP'10, LNCS 6012*, pages 287–306. Springer, 2010.

[19] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL'03*, pages 185–197. ACM Press, 2003.

[20] J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 162–169, New York, NY, USA, 2006. ACM.

[21] J. J. Hunt, I. Tonin, and F. B. Siebert. Using global data flow analysis on bytecode to aid worst case execution time analysis for real-time java programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 97–105, New York, NY, USA, 2008. ACM.

[22] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual. Draft Revision 1.200*, Feb. 2007.

[23] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

[24] M. Montenegro, O. Shkaravska, M. van Eekelen, and R. Peña. Interpolation-based height analysis for improving a recurrence solver. In *Proceedings of the 2nd Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2011*, LNCS 7177, pages 36–53. Springer, 2012.

[25] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 465–486. Springer Berlin / Heidelberg, 2004.

[26] E. Poll, P. Chalin, D. Cok, J. Kiniry, and G. T. Leavens. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, volume 4111 of LNCS*, pages 342–363. Springer, 2006.

[27] O. Shkaravska, R. Kersten, and M. Van Eekelen. Test-based inference of polynomial loop-bound functions. In A. Krall and H. Mössenböck, editors, *PPPJ'10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, ACM Digital Proceedings Series, pages 99–108, 2010.

[28] O. Shkaravska, M. van Eekelen, and R. van Kesteren. Polynomial size analysis of first-order shapely functions. *Logic in Computer Science*, 2:10(5), 2009.

[29] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial Size Analysis for First-Order Functions. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications (TLCA'2007), Paris, France*, volume 4583 of *LNCS*, pages 351–366. Springer, 2007.

[30] F. Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. PhD thesis, University of Karlsruhe, 2002.

[31] R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In *16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07), Paris, France*, volume 216C of *Electronic Notes in Theoretical Computer Science*, pages 45–63, 2008.

[32] S. Wang, Z. Qiu, S. Qin, and W.-N. Chin. Stack bound inference for abstract java bytecode. In *Proceedings of the 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE '10, pages 57–66, 2010.

[33] G. Wedzinga and K. Wiegmink. Using CHARTER tools to develop a safety-critical avionics application in Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES'12)*, 2012. To appear.

[34] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.