

Test-Based Inference of Polynomial Loop-Bound Functions*

Olha Shkaravska Rody Kersten

Radboud University Nijmegen
{shkarav,r.kersten}@cs.ru.nl

Marko van Eekelen

Radboud University Nijmegen and Open Universiteit
marko@cs.ru.nl

Abstract

This paper presents an interpolation-based method of inferring arbitrary degree loop-bound functions for Java programs. Given a loop, by its “loop-bound function” we mean a function with the numeric program variables as its parameters, that is used to bound the number of loop-iterations. Using our analysis, loop-bound functions that are polynomials with natural, rational or real coefficients can be found.

Analysis of loop bounds is important in several different areas, including worst-case execution time (WCET) and heap consumption analysis, optimising compilers and termination-analysis. While several other methods exist to infer numerical loop bounds, we know of no other research on the inference of non-linear loop-bound *functions*. Additionally, the inferred bounds are provable using external tools, e.g. KeY.

To infer a loop-bound function for a given loop it is instrumented with a counter and executed on a *well-chosen* set of values of the numerical program variables. By well-chosen we mean that using these test values and the corresponding values of the counter, one can construct a unique interpolating polynomial. The uniqueness and the existence of the interpolating polynomial is guaranteed if the input values are in the so-called NCA-configuration, known from multivariate-polynomial interpolation theory. The constructed interpolating polynomial presumably bounds the dependency of the number of loop iterations on arbitrary values of the program variables. This hypothesis is verified by a third-party proof assistant.

A prototype tool has been developed which implements this method. This prototype can infer piecewise polynomial loop-bound functions for a large class of loops in Java programs. Applicability of the prototype has been tested on a series of safety-critical case studies. For most of the loops in the case studies, loop-bound functions could be inferred (and verified using a proof assistant).

Categories and Subject Descriptors F.3.1 [LOGICS AND MEANINGS OF PROGRAMS]: Specifying and Verifying and Reasoning about Programs; D.2.4 [SOFTWARE ENGINEERING]: Software/Program Verification

General Terms Verification, Performance, Reliability

* This work was funded by the Artemis Joint Undertaking in the CHARACTER project, grant-nr. 100039, and the Laboratory for Quality Software (LaQuSo).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPPJ '10, September 15–17, 2010, Vienna, Austria.
Copyright © 2010 ACM 978-1-4503-0269-2...\$10.00

Keywords Loop-bound function inference, Program verification, Polynomial interpolation, NCA-configuration, Termination, JML

1. Introduction

Loop-bounds are important in several different fields. The most obvious are worst-case execution time (WCET) and heap consumption analysis. Both are key issues for safety-critical systems, e.g. automotive and avionics applications. Compiler optimisations that transform loops, e.g. loop-unrolling, may also depend on knowledge about loop-bounds. Furthermore, termination of a program can only be proved if an upper bound on the number of loop iterations exists for each loop in the program.

In the presented paper we describe test-based inference of piecewise polynomial loop-bound functions for Java loops with first-order numerical loop conditions. Given a program with a loop, the corresponding *loop-bound function* (LBF) expresses an upper bound on the number of the loop iterations depending on (some of) the numerical program variables and data sizes. A similar term that is often used is “ranking function”. However, because our definition is more specialised (we consider only loops) and to avoid confusion with the term of the same name used in the information retrieval community, we prefer the term “loop-bound function”.

Consider for example the loop in Listing 1. While several other methods exist that are able to infer a numerical loop bound for certain values of *i*, for instance minimal or maximal values, we infer the loop-bound *function* $15-i$, which can be used to bound (or in this case, calculate exactly) the number of iterations of this loop for *arbitrary* values of *i*. Only one other paper [9] is known to the authors where symbolic bounds are inferred, but there only a limited use of non-linear terms in bounds is possible. In other articles, soundness is not usually discussed, but the correctness of the LBF inferred by the presented method can be checked by external tools. By proving correctness of the bound, termination of the loop is also proved inherently.

```
1 while (i < 15) {  
2   i++;  
3 }
```

Listing 1. A typical single (i.e. not nested) while-loop.

The schema of our approach is as follows. Given a loop, it is first placed into a testing method. The loop in this method is instrumented with a counter and this testing method outputs the number of iterations of the loop on given values of the program variables. In this paper, we take into consideration numerical program variables that occur in the loop condition and its body. Then, the method is executed on a *well-chosen* set of inputs, which we call test values or, sometimes, following polynomial-interpolation terminology, test nodes. By well-chosen set of test values we mean that using these test values and the corresponding values of the counter, one can construct a unique interpolating polynomial. The uniqueness and

the existence of the interpolating polynomial is guaranteed if the input values are in the so-called NCA-configuration, known from multivariate-polynomial interpolation theory. This issue is highlighted in Section 2, explaining application of polynomial interpolation. The obtained data are used to compute the corresponding interpolating polynomial. The inferring part of the procedure outputs a method, annotated with the inferred bound, e.g. annotated in JML [15] by the `decreases`-expression. The correctness of the annotation is verified by an external checking tool. In our prototype implementation the method with the annotated loop is run through KeY [3], which contains a verification-condition generator and a theorem prover.

To obtain concrete loop bounds (i.e. the concrete number of iterations on concrete data), the obtained LBF may be applied to the results of data-flow analysis, possibly accelerated by abstract interpretation and/or program slicing. Several examples exist in the literature [7, 14] indicating how this might be implemented.

This work builds on our previous research on size analysis [17], where we studied polynomial dependencies of the sizes of output data structures (e.g. the length of a linked list) on the sizes of input data structures. A similar algorithm as is used here for generating loop bounds was used in [18] to infer size relations.

The running example throughout this paper is a while-loop with a quadratic LBF, given in Listing 2. This bound is successfully inferred by the prototype implementation and can be verified using KeY.

```

1 while (x>0 && i>0 && i<x && j>0 && j<=x) {
2   if (j==x) { i++; j = 0; }
3   j++;
4 }

```

Listing 2. A single loop with the quadratic LBF $x^2 - xi - j + 1$.

This research is conducted in the context of the Critical and High Assurance Requirements Transformed through Engineering Rigour (CHARTER) project¹. The goal of this project is to ease, accelerate, and cost-reduce the certification of safety-critical embedded systems by melding realtime Java, Model Driven Development, rule-based compilation, and formal verification. It will be part of a larger chain of tools developed in this project.

We recapitulate polynomial interpolation in Section 2. The LBF inference method is introduced in Section 3. We then briefly discuss the prototype implementation and a series of case studies in Section 4. Extensions to the basic method, that serve to handle more complex cases, are discussed in Section 5. The method is evaluated in Section 6. Related work is discussed in Section 7. We outline future work in Section 8 and conclude the paper in Section 9.

2. 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 [5].

For 1-variable interpolation this condition is well-known: all the test nodes must be different. Recapitulate it in more detail. A polynomial $p(z)$ of degree d with coefficients a_0, \dots, a_d can be written as follows:

$$a_0 + a_1 z + \dots + a_d z^d = p(z)$$

The values of the polynomial function in any pairwise different $d + 1$ points determine a system of linear equations w.r.t. the polynomial coefficients. More specifically, given the set $(z_i, p(z_i))$

of pairs of numbers, where $0 \leq i \leq d$, and coefficients a_0, \dots, a_d , the system of equations can be represented in the following matrix form, where only the a_i are unknown:

$$\begin{pmatrix} 1 & z_0 & \dots & z_0^{d-1} & z_0^d \\ 1 & z_1 & \dots & z_1^{d-1} & z_1^d \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & z_{d-1} & \dots & z_{d-1}^{d-1} & z_{d-1}^d \\ 1 & z_d & \dots & z_d^{d-1} & z_d^d \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \\ a_d \end{pmatrix} = \begin{pmatrix} p(z_0) \\ p(z_1) \\ \vdots \\ p(z_{d-1}) \\ p(z_d) \end{pmatrix}$$

The determinant of the matrix is called a *Vandermonde* determinant. For pairwise different points z_0, \dots, z_d it is non-zero. This means that, as long as the output values $p(z_i)$ are known for $d + 1$ different inputs z_i , there exists a unique solution for the system of equations and, thus, a unique interpolating polynomial.

The condition under which there exists a unique *multivariate* polynomial $p(z_1, \dots, z_k)$ that interpolates multivariate data is not trivial. Using the result from [5], we have shown how to generate test data for size analysis of *functional programs* in [17]. Here we recall the basic facts from these papers. First, a polynomial $p(z_1, \dots, z_k)$ of a degree d and dimension k (the number of variables) has $N_d^k = \binom{d+k}{k}$ coefficients. Let a set of values f_i of a real function f be given and let \bar{z} denote a vector-variable (z_1, \dots, z_k) . A set $W = \{\bar{w}_i = (z_{i1}, \dots, z_{ik}) : i = 1, \dots, N_d^k\}$ of points in a real k -dimensional space forms the set of *interpolation nodes* if there is a unique polynomial $p(\bar{z}) = \sum_{0 \leq j_1 + \dots + j_k \leq d} a_{j_1 \dots j_k} z_1^{j_1} \dots z_k^{j_k}$ with the total degree d with the property $p(\bar{w}_i) = f_i$, where $1 \leq i \leq N_d^k$. In this case one says that the polynomial p interpolates the function f at the nodes \bar{w}_i . The condition on W , which assures the existence and uniqueness of an interpolating polynomial, is geometrical: it describes a node configuration, called *Node Configuration A*, **NCA** for short, [5], in which the nodes from W should be placed in \mathcal{R}^k . The multivariate Vandermonde determinant computed from such points is non-zero. Thus, the corresponding system of linear equations w.r.t. the polynomial's coefficients has a unique solution. For a two-dimensional polynomial of degree d , the condition on the nodes that guarantees a unique polynomial interpolation is as follows:

N_d^2 nodes forming a set $W \subset \mathcal{R}^2$ lie in a 2-dimensional NCA if there exist lines $\gamma_1, \dots, \gamma_{d+1}$ 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 $\gamma_d \setminus \gamma_{d+1}, \dots$, and finally 1 node of W lies on $\gamma_1 \setminus (\gamma_2 \cup \dots \cup \gamma_{d+1})$.

A typical instance of such a configuration is a 2-dimensional *grid*. An example of a two-dimensional grid based on integers is given in Figure 1.

For dimensions $k > 2$ the 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 some N_{d-1}^{k-1} of the given nodes lying in $(k - 1)$ -dimensional NCA for the degree d ,
- there is a $(k - 1)$ -dimensional hyperplane such that it contains some N_{d-1}^{k-1} nodes, lying in $(k - 1)$ -dimensional NCA for the degree $d - 1$, and these nodes do not lie on the previous hyperplane,
- in general, for any $0 \leq i \leq d$, there is a $(k - 1)$ -dimensional hyperplane such that it contains some 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.

¹<http://charterproject.ning.com/>

For instance, for the example in Listing 2, we might assume that the loop-bound function is a quadratic polynomial (so, $d = 2$) and depends on three variables, x , i and j . Recall, that a quadratic function of three variables has $\binom{5}{3} = 10$ coefficients: $p(x, i, j) = a_{200}x^2 + a_{020}i^2 + a_{002}j^2 + a_{110}xi + a_{101}xj + a_{011}ij + a_{100}x + a_{010}i + a_{001}j + a_{000}$. Therefore we need 10 three-dimensional points in \mathcal{R}^3 -NCA. According to the definition above we need:

- A set of $\binom{2+2}{2} = 6$ points on a plane in \mathcal{R}^2 -NCA: we take the hyperplane $j = 1$ and the following nodes:

x	i	j
2	1	1
3	1	1
4	1	1
3	2	1
4	2	1
4	3	1

These points are given (projected on the hyperplane $j = 1$) in Figure 1.

- A set of $\binom{2+1}{2} = 3$ points in \mathcal{R}^2 -NCA on another plane: we take the hyperplane $j = 2$ and nodes

x	i	j
3	1	2
4	1	2
3	2	2

- A single $\binom{2+0}{2} = 1$ point on yet another plane in \mathcal{R}^2 -NCA: we take just $(x = 4, i = 1, j = 3)$.

The corresponding system of linear equations is:

$$\left\{ \begin{array}{l} 2^2 a_{200} + 1^2 a_{020} + 1^2 a_{002} + 2a_{110} + 2a_{101} + 1a_{011} + 2a_{100} + 1a_{010} + 1a_{001} + 1a_{000} = 2 \\ 3^2 a_{200} + 1^2 a_{020} + 1^2 a_{002} + 3a_{110} + 3a_{101} + 1a_{011} + 3a_{100} + 1a_{010} + 1a_{001} + 1a_{000} = 6 \\ 4^2 a_{200} + 1^2 a_{020} + 1^2 a_{002} + 4a_{110} + 4a_{101} + 1a_{011} + 4a_{100} + 1a_{010} + 1a_{001} + 1a_{000} = 12 \\ 3^2 a_{200} + 2^2 a_{020} + 1^2 a_{002} + 3 \cdot 2a_{110} + 3a_{101} + 2a_{011} + 3a_{100} + 2a_{010} + 1a_{001} + 1a_{000} = 3 \\ 4^2 a_{200} + 2^2 a_{020} + 1^2 a_{002} + 4 \cdot 2a_{110} + 4a_{101} + 2a_{011} + 4a_{100} + 2a_{010} + 1a_{001} + 1a_{000} = 8 \\ 4^2 a_{200} + 3^2 a_{020} + 1^2 a_{002} + 4 \cdot 3a_{110} + 4a_{101} + 3a_{011} + 4a_{100} + 3a_{010} + 1a_{001} + 1a_{000} = 4 \\ 3^2 a_{200} + 1^2 a_{020} + 2^2 a_{002} + 3a_{110} + 3 \cdot 2a_{101} + 2a_{011} + 3a_{100} + 1a_{010} + 2a_{001} + 1a_{000} = 5 \\ 4^2 a_{200} + 1^2 a_{020} + 2^2 a_{002} + 4a_{110} + 4 \cdot 2a_{101} + 2a_{011} + 4a_{100} + 1a_{010} + 2a_{001} + 1a_{000} = 11 \\ 3^2 a_{200} + 2^2 a_{020} + 2^2 a_{002} + 3 \cdot 2a_{110} + 3 \cdot 2a_{101} + 2 \cdot 2a_{011} + 3a_{100} + 2a_{010} + 2a_{001} + 1a_{000} = 2 \\ 4^2 a_{200} + 1^2 a_{020} + 3^2 a_{002} + 4a_{110} + 4 \cdot 3a_{101} + 3a_{011} + 4a_{100} + 1a_{010} + 3a_{001} + 1a_{000} = 10 \end{array} \right.$$

Its solution is $(1, 0, 0, -1, 0, 0, 0, 0, -1, 1)$, which yields the polynomial $p(x, i, j) = x^2 - xi - j + 1$.

3. Inference of Loop-Bound Functions

Our method is designed for loops with conditions in the form of propositional logic expressions over numerical (in)equalities. Formally:

$$C := sC \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \\ sC := e_1 [<, >, \leq, \geq, =, \neq] e_2$$

where e_i are arithmetical expressions.

For now, we limit our focus to loops where the conditions are conjunctions over linear (in)equalities. The analysis of loops where

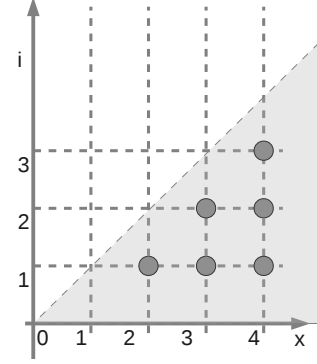


Figure 1. An example of well-chosen test nodes for a polynomial $g(x, i) = a_{20}x^2 + a_{11}xi + a_{02}i^2 + a_{10}x + a_{01}i + a_{00}$. These may be used to reconstruct a polynomial $g(x, i) = p(x, i, 1)$, where p is the polynomial bound for our running example. The grey area represents points that satisfy the condition $x < i$.

the condition contains disjunctions is discussed in Section 5. Note that the limiting the loop-conditions to linear (in)equalities does not mean limiting to linear LBFs. The loop in Listing 2 has a non-linear LBF for instance.

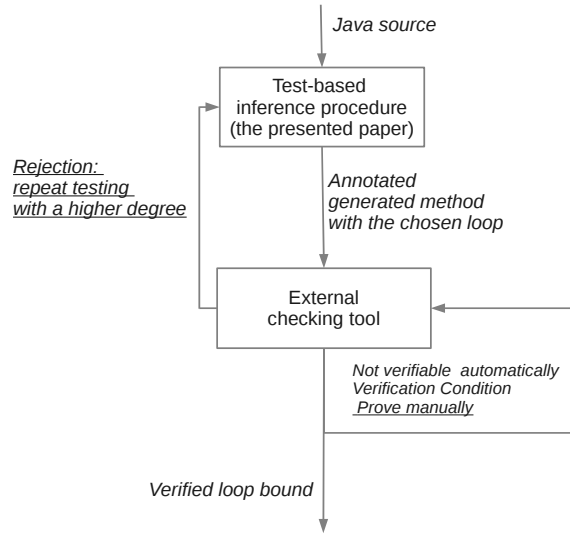


Figure 2. Test-based procedure from a helicopter view: infer-and-check cycle.

In Figure 2 we give a helicopter view of the test-based infer-and-check procedure. First, a user inputs the Java source code to the inference procedure. Then the procedure makes a hypothesis of a LBF, based on test-runs. This hypothesis is expressed in some conventional annotations, like JML, so the annotated method output can be read by an external checker that checks if the inferred bound is correct. Manual steps might be necessary to construct the proof. If the user concludes that such a proof cannot be found, (s)he might go back to the inference procedure and try again with a higher degree of a polynomial LBF.

In Figure 3, we zoom in on the test-based inference module. We start with Java source code and pick a loop for which one wants to infer an LBF. The loop is (automatically) inserted in a new method and instrumented with a counter, which is returned at the end of the method. The parameters of the method are the

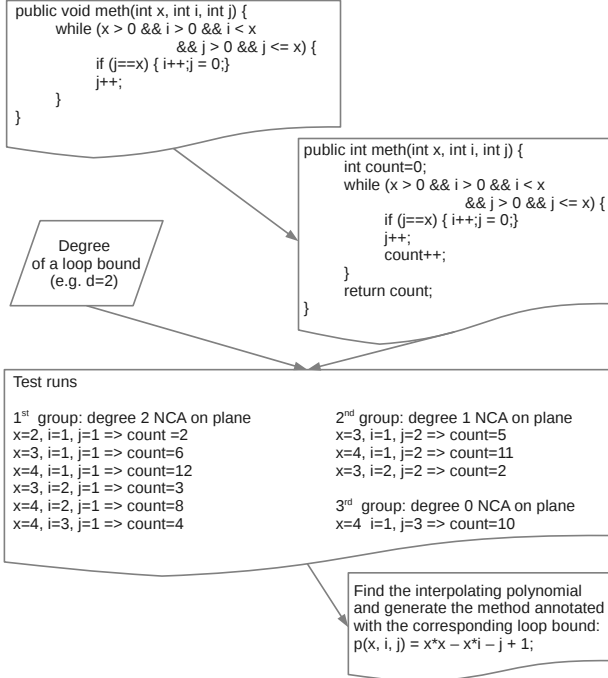


Figure 3. Test-based inference module in more detail. The choice of test nodes is explained in Section 2.

numerical variables that occur in the loop condition and in its body. In Section 8.2, we discuss a combination with program slicing techniques, which would yield exactly the variables on which the loop bound depends. The new method is now executed for a given degree and an appropriate set of values of these parameters, i.e. on so called *test nodes*. For instance, in our running example ($x = 2, i = 1, j = 1$) is an admissible test node. A well-chosen complete set of test nodes for this loop is given in the figure. The set consists of 10 nodes, since a polynomial of degree 2 of 3 variables has 10 coefficients: $p(x, i, j) = a_{200}x^2 + a_{020}i^2 + a_{002}j^2 + a_{110}xi + a_{101}xj + a_{011}ij + a_{100}x + a_{010}i + a_{001}j + a_{000}$. The result of a test run is the number of iterations for the corresponding node. For instance, with $(x = 2, i = 1, j = 1)$ the loop body is executed 2 times, so the test method returns `count = 2`. From the results of the test-runs a polynomial over the parameters can be calculated which interpolates the test results.

Multiple tactics are possible to guess the degree of the polynomial. It can be left to the user to supply it as input to the procedure, or an increasing degree can be tried, up to a certain bound. When a degree that is too low is supplied, the method will still find an LBF, but the checker will reject it. When a degree that is too high is given, the right polynomial will still be found, but more test-runs are needed.

3.1 LBF Inference: The Basic Method

This polynomial interpretation method was already applied to Size Analysis in previous work [17, 18]. The *main challenge* we face when we adjust the interpolation theory to inferring imperative loop-bound functions is that test data must not only lie on a grid (or more generally, be in NCA), but also satisfy the loop condition C . In Figure 1 we show the set of points satisfying the (in)equalities $i < x, i > 0$ and $x > 0$. This corresponds to the loop condition in our running example for the fixed $j = 1$. Whenever the loop condition is violated the loop is not executed and the testing method, which is wrapped around the loop, outputs 0. Therefore, if we con-

structed the interpolation polynomial using a node(s) that does not satisfy the loop condition, we would obtain for sure an incorrect loop bound.

The problem of generating test data for imperative loops is formalised as follows:

- given:
 - a degree d ,
 - the number of variables k , on which the loop-bound function depends,
 - a loop condition C ,
- to find N_d^k nodes in NCA that satisfy C .

We have reduced this task to the following one: *construct an integer grid in \mathcal{R}^k , such that it is based on $d + 1$ parallel hyperplanes and contains N_d^k nodes, where*

- there are some N_d^{k-1} nodes in $(k - 1)$ -dimensional NCA for the degree d that lie on one of the hyperplanes and satisfy the corresponding projection of C to this hyperplane,
- there are some N_{d-1}^{k-1} nodes in $(k - 1)$ -dimensional NCA for the degree $d - 1$ that lie on another hyperplane and satisfy the projection of C on this hyperplane,
- there are some N_{d-i}^{k-1} nodes in $(k - 1)$ -dimensional NCA for the degree $d - i$ that lie on a fresh hyperplane and satisfy the projection of C on this hyperplane, $0 \leq i \leq d$,
- and the remaining 1 node lies on the remaining hyperplane and satisfying the corresponding projection of C .

In general terms, our approach is based on search of the appropriate nodes on hyperplanes $x_1 = i_0, \dots, i_d$. The search is inductive on the number of variables k . To bound the search space one uses an external optimisation procedure solving tasks of the form $f(x_1, \dots, x_k) \rightarrow \min$, where x_1, \dots, x_k satisfy the constraints $C(x_1, \dots, x_k)$. Currently in our prototype we use a linear programming solver, and therefore, the prototype handles only linear loop conditions. In general, one may use non-linear optimisation software, such as the implementation of the Augmented Lagrangian Genetic Algorithm (ALGA) by MathWorks² or the open-source Java package Sigoa³.

The rest of this subsection is structured as is the inference procedure: generating test-nodes, conducting the tests and interpolating a polynomial LBF.

3.1.1 The algorithm for generating test-nodes

1. Run the chosen optimisation procedure for the objective functions $x_i \rightarrow \min, x_i \rightarrow \max$ and the constraints constituted from the loop conditions and the additional bounds $m_i \leq x_i \leq M_i$, where m_i, M_i are predefined resp. minimal and maximal admissible values of the variables x_i , with $1 \leq i \leq k$.
The results define the k -dimensional box, that bounds the set defined by C (within the minimal-maximal values). We only look for nodes inside this box, because we know that others do not satisfy the loop condition.
2. Obtain search hyperplanes H_j by cutting the bounding box on d congruent “slices”, $j = 0, \dots, d$.
3. Amongst these hyperplanes, search for one that contains N_d^{k-1} nodes in $(k - 1)$ -dimensional NCA for the degree d and the projection of C on this hyperplane, etc. as explained above.

²<http://www.mathworks.com/access/helpdesk/help/toolbox/gads/bqf8bdd.html>

³<http://sigoa.sourceforge.net/>

4. If the search succeeds, then stop. Otherwise, refine the grid by increasing the number of hyperplanes (e.g. by decreasing the distance between them) and repeat the search for the refined grid.

This procedure finds test-nodes that both satisfy the loop condition and lie in NCA, if they exist on a grid within the minimal-maximal values m_i, M_i . It finds suitable test-nodes for the case-study examples. To refine the search algorithm, one may add other, than rectangular grids, kinds of NCA, like e.g. *pencil* configurations.

3.1.2 Run tests

When suitable test nodes have been selected, we can now run the tests. Of course, because the investigated loops are actually executed, termination of the inference procedure depends on termination of those loops. The LBF inference procedure terminates *if* the considered loop terminates for all inputs.

Assuming that infinite loops are undesirable in general, but especially for loops for which one seeks to bound the number of iterations, “finding” non-termination for certain inputs is a valuable result in itself. An implementation can never conclude non-termination, but it may quit execution after a particular amount of time has passed and hint the user that there is a large chance that the loop does not terminate on the considered inputs.

3.1.3 Find the interpolating polynomial

When all the tests have produced iteration counts (i.e. all have terminated), then we can now fit a polynomial, which interpolates these results. Because the test-nodes satisfy NCA, we know that a single interpolating polynomial exists.

3.2 Dealing with LBFs with rational or real coefficients

As stated earlier, LBFs can be 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 iteration must be. Consider for instance the loop in Listing 3.

```
1 while (start < end) {
2   start += 4;
3 }
```

Listing 3. 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 $\lceil \frac{\text{end}-\text{start}}{4} \rceil$. In other words, when $\text{end} - \text{start}$ does not equate to a natural number, for instance to $\frac{3}{4}$, it must be *ceiled*. Generally, when the coefficients of the polynomial LBF $p(\bar{z})$ are not natural numbers, the actual bound should be read as $\lceil p(\bar{z}) \rceil$. When the coefficients are natural, we omit the ceiling notation.

3.3 Expressing the LBF in JML

In this section we discuss how we can express the found LBF in JML, in order to be verified by an external tool.

The result of our method is Java code annotated with JML, in which the inferred LBF is expressed. Loop-bound functions are most easily 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 remains greater than or equal to 0, see the JML reference manual [13]. It therefore forms an upper-bound on the number of iterations.

We want an external tool to verify the LBF for the case where the loop condition initially holds, otherwise the `decreases`-clause is not guaranteed to be ≥ 0 initially (and the loop will iterate exactly 0 times). Therefore, the loop condition is added as a precondition

to the constructed method. The example from Listing 2 is shown in annotated form in Listing 4.

```
1 /*@
2   requires x>0 && i>0 && i<x && j>0 && j<=x;
3   ensures true;
4 */
5 public void meth(int x, int i, int j) {
6
7   //@ assignable i,j;
8   //@ loop_invariant true;
9   //@ decreases x*x - x*i - j + 1;
10  while (x>0 && i>0 && i<x && j>0 && j<=x) {
11    if (j==x) { i++;j = 0;}
12    j++;
13  }
14 }
```

Listing 4. The inferred LBF for the example in Listing 2 expressed as a JML annotation.

Unfortunately, there is no ceiling operation available in JML. We therefore have to overestimate the bound slightly when the LBF has rational or real coefficients. Since $\lceil p(\bar{z}) \rceil \leq p(\bar{z}) + 1$, we can do this safely by adding one instead of ceiling.

Although our method works for both integer and floating-point numbers, and JML can handle them, no verification tools for Java programs annotated with JML are known to the authors that have support for floating-point numbers.

3.4 Complexity: exponential w.r.t. the number of variables k

The first sub-procedure in the presented inference method is an external optimisation procedure used to bound the test-nodes search space. Typically, the complexity of optimisation methods depends on the number of (in)equations in the constraints, number of variables (the space’s dimension) and complexity of (in)equations. For non-linear constraints the worst-case complexity is, as a rule, exponential, but one often uses “smart search” algorithms providing better average computation time. For instance, in genetic algorithms the search is directed by e.g. the value of a penalty function that decreases when one searches in the “right direction”.

For the remaining parts of the inference method we can give independent estimations of complexity. These parts are:

- the search of test nodes that, as one intuitively expects, has the most significant complexity, which we will discuss right now, below,
- the runs ($N_d^k = \binom{d+k}{k}$ times) of the test method on the test nodes,
- solving a system of N_d^k linear equations w.r.t. N_d^k variables that has the worst complexity $O((N_d^k)^3)$; with some advanced matrix-multiplication algorithms the complexity may be between $O((N_d^k)^2)$ and $O((N_d^k)^3)$.

Searching of test nodes is the most time-consuming part of the inference procedure (besides, probably, non-linear optimisation part). Let $\mathcal{N}(d, k)$ denote the time for finding the nodes for a polynomial of the degree d with k variables. Consider its behaviour from the best to the worst case, with $\mathcal{N}_{\min}(d, k)$ denoting the best computation time.

In the *best case* we just cut the k -dimensional cube by $d + 1$ hyperplanes of the dimension $k - 1$, and find immediately N_{d-i}^{k-1} points on the i -th hyperplane in time $\mathcal{N}_{\min}(d - i, k - 1)$, where $0 \leq i \leq d$. Therefore, we may assume that $\mathcal{N}_{\min}(d, k - 1) = \mathcal{N}_{\min}(d, k - 1) + \mathcal{N}_{\min}(d - 1, k - 1) + \dots + \mathcal{N}_{\min}(1, k - 1) + \mathcal{N}_{\min}(0, k - 1) + (d + 1)$ that includes the time for $d + 1$ recursive calls. We can show by induction on k that $\mathcal{N}_{\min}(d, k) = O\left(\frac{d^k}{k!}\right)$.

Indeed, for $k = 1$ we have to pick up $d + 1$ different points on the line, so $\mathcal{N}_{\min}(d, 1) = d + 1 = O(\frac{d}{1})$. For $k = 2$ we have $\mathcal{N}_{\min}(d, 2) = (d + 1) + d + \dots + 1 + (d + 1) = \frac{d(d+1)}{2} + (d + 1) = O(\frac{d^2}{2})$. Using the induction assumption, $\mathcal{N}_{\min}(d, k) = \sum_{i=0}^d O(\frac{(d-i)^{k-1}}{(k-1)!}) + (d+1) = O(\frac{1}{(k-1)!} \sum_{i=0}^d j^{k-1}) + (d+1) \approx O(\frac{1}{(k-1)!} \int_0^d x^{k-1} dx) + (d+1) = O(\frac{d^k}{k!})$.

In the “middle” case the initial collection of $(d+1)$ hyperplanes does have all the points in the necessary configuration, but, roughly, one has to reorder hyperplanes to get the k -dimensional NCA configuration. That is, the $i = 0$ -th hyperplane does not contain enough, i.e. N_d^{k-1} , $(k-1)$ -dimensional points, so in general we have to look through all $d+1$ hyperplanes. Next, for N_{d-1}^{k-1} points we have to search in d remaining hyperplanes, etc. So, for N_{d-i}^{k-1} points we search in $d+1-i$ hyperplanes. Therefore, $\mathcal{N}(d, k) = \sum_{i=0}^d ((d+1-i)(\mathcal{N}(d-i, k-1) + 1))$, including the recursive calls (with “+1” staying for the recursive call of the procedure for $d-i, k-1$). Then, the estimate is

$$\begin{aligned} \mathcal{N}(d, k) &\leq \sum_{i=0}^d ((d+1-i)(\mathcal{N}(d, k-1) + 1)) = \\ &(\mathcal{N}(d, k-1) + 1) \sum_{i=0}^d (d+1-i) = \\ &(\mathcal{N}(d, k-1) + 1) O(\frac{d^2}{2}) \leq \\ &(\mathcal{N}(d, k-2) + 1) O(\frac{d^4}{4}) + O(\frac{d^2}{2}) = \\ &O((\frac{d^2}{2})^k) \end{aligned}$$

Now, it is clear that the *the worst-case* computation time of node search is exponential in k . Different versions of the search procedure provide different bases of the exponent or differ by a multiple, that may be quite large. Here we consider one of the versions (implemented in the prototype) with accelerated generation of new collections of hyperplanes. In the worst case, if we fail to find enough nodes w.r.t. the current collection of hyperplanes, we have to generate another collection of $D > d+1$ hyperplanes for a refined grid. Similarly to the estimates above, the estimate is $\mathcal{N}(d, k) = \sum_{i=0}^d (D+1-i)(\mathcal{N}(d-i, k-1) + 1) \leq (\mathcal{N}(d, k-1) + 1) O(\frac{D^2}{2}) = O((\frac{D^2}{2})^k)$. After failing with the first hyperplane collection, D takes consecutively the values $2(d+1)$, $2^3 2(d+1)$, \dots , $2^{8i+1}(d+1)$, with $0 \leq i \leq i_{\max}$ and for i_{\max} the following holds. It is such that $2^{8i_{\max}+1}(d+1) \leq M+1$, where M is the (length of the) side of the bounding box, generated by the optimisation procedure on the first step. So, we obtain that $i_{\max} \leq \frac{1}{8}(\log_2 \frac{M+1}{d+1} - 1)$. The worst-case time, when we have to go through all the possible cuts, is then

$$\begin{aligned} \sum_{i=0}^{i_{\max}} \left(\left(\frac{(2^{8i+1}(d+1))^2}{2} \right)^k \right) &= \\ O(2^k (d+1)^{2k}) \sum_{i=0}^{i_{\max}} O(2^{16k})^i &= \\ O(2^k (d+1)^{2k} \frac{(2^{16k})^{i_{\max}+1} - 1}{2^{16k} - 1}) & \end{aligned}$$

Taking into account the estimate for i_{\max} we obtain that $\mathcal{N}(d, k)$ does not exceed

$$O\left(\left(\frac{2}{2^{16}} (d+1)^2 \right)^k \left(\frac{M+1}{2(d+1)} \right)^{2k} \right) = O\left(\left(\frac{1}{2^{17}} (M+1)^2 \right)^k \right)$$

4. Prototype and Case Studies

We have created a prototype implementation of the method in Java. This prototype can be used to load Java source files, select a loop to analyse, input an expected degree, infer a loop-bound function (LBF) and output Java code containing JML annotations in order

to prove this inferred LBF using an external tool, for instance KeY [3] or ESC/Java2 [15].

For the prototype, existing software packages were used as much as possible, for instance for bounding the test-node search space and for solving the interpolation matrix. Around 3000 lines of code were added to create a working prototype, including a graphical user interface.

JML annotations can be generated for all of the loops listed in this paper. We were able to prove all the inferred LBFs using KeY. Additionally, we have conducted three case studies of safety-critical Java systems, suggested as test cases by the CHARTER partners.

- **Collision detector case study from [11].** The first case is the collision detector example from the paper “Provable Correct Loop bounds for Realtime Java Programs” by James Hunt et al [11]. 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)⁴. The package is described in detail in [16].
- **CDx Collision Detector package.** The CDx Collision Detector package⁵ is a publicly available Real-Time Java Benchmark. It is described in [12].

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

Table 1. Summary of the cases studied.

The results are shown in Table 1. 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 an LBF for these loops using our prototype and prove it using KeY. All of the found LBFs were linear, i.e. of degree one.

In the case studies, apparently, enough test-nodes are found after just a few cuts of the k -dimensional search space. This leads us to believe that the average complexity of the method lies somewhere around $O(D^{2k})$ for $D = 2^9(d+1)$, rather than near the worst-case complexity. For the examples in the case studies this amounts to approximately one second spent in LBF inference. KeY was able to prove all the LBFs fully automatically, for which it requires approximately 5 to 10 seconds.

5. Extensions

Two extensions were made to the core method in order to deal with a greater class of loops. The first extension enables analysis of loops with disjunctions in their conditions. The second extension enables analysis of loops with conditionals (if-statements) inside their bodies.

⁴<http://diana.skysoft.pt/>

⁵<http://adam.lille.inria.fr/soleil/rcd/>

5.1 Piecewise LBFs for disjunctive loop conditions

Here, we consider loop conditions in disjunctive normal form (DNF) over arithmetical (in)equalities:

$$\bigvee_{i=1}^n \left(\bigwedge_{j=1}^{m_i} (e_{lij} \mathbf{b} e_{rij}) \right)$$

with $\mathbf{b} \in \{<, >, =, \neq, \leq, \geq\}$. Note that any expression in propositional logic can easily be converted into DNF, using the laws of distribution and De Morgans theorems.

We transform the DNF loop condition into a DNF in which the conjuncts represent pairwise non-overlapping numeric sets. We use the fact that:

$$B_1 \vee \dots \vee B_n = \bigvee_{I \subseteq \{1, \dots, n\}, I \neq \emptyset} \left(\bigwedge_{i \in I} B_i \wedge \bigvee_{j \in \bar{I}} B_j \right)$$

where \bar{I} denotes the complement of I in $\{1, \dots, n\}$. Formally, the *disjunction-splitting* operation for e.g. a two-conjunct DNF $\bigwedge_{i=1}^{m_1} A_{1i} \vee \bigwedge_{i=1}^{m_2} A_{2i}$, where each A_{ji} is an inequation, is defined by

$$\text{split}(\bigwedge_{i=1}^{m_1} A_{1i} \vee \bigwedge_{i=1}^{m_2} A_{2i}) := \bigwedge_{i_1=1, i_2=1}^{m_1, m_2} A_{1i_1} \wedge (\neg A_{2i_2}) \vee \bigwedge_{i_1=1, i_2=1}^{m_1, m_2} A_{1i_1} \wedge A_{2i_2} \vee \bigwedge_{i_1=1, i_2=1}^{m_1, m_2} (\neg A_{1i_1}) \wedge A_{2i_2}$$

Together, the disjunctive conjuncts of a loop condition determine a *piecewise* LBF.

Consider the example given in Listing 5.

```
1 while (start < end && (end < 40 || end > 100))
2   start++;
```

Listing 5. While-loop with disjunctions and two parameters in its condition.

The procedure first splits up the loop condition into three disjunctive parts $(\text{start} < \text{end}) \wedge (\text{end} < 40) \wedge \neg(\text{end} > 100)$, $(\text{start} < \text{end}) \wedge (\text{end} > 100) \wedge \neg(\text{end} < 40)$ and $(\text{start} < \text{end}) \wedge (\text{end} < 40) \wedge (\text{end} > 100)$. Since $(\text{end} < 40)$ and $(\text{end} > 100)$ cannot both be true, this can easily be simplified to the following two pieces: $(\text{start} < \text{end}) \wedge (\text{end} < 40)$ and $(\text{start} < \text{end}) \wedge (\text{end} > 100)$

Then it constructs test methods for two separate loops, one with each condition. From this point on, the regular inference procedure runs. It generates the following piecewise bound:

$$\begin{cases} \text{end} - \text{start} & \text{if } (\text{start} < \text{end}) \wedge (\text{end} < 40) \\ \text{end} - \text{start} & \text{if } (\text{start} < \text{end}) \wedge (\text{end} > 100) \\ 0 & \text{else} \end{cases}$$

In this case the polynomials in both clauses of the disjunction coincide.

Disjunction-splitting takes care of loop conditions containing disjunctions, as long as the body of the loop satisfies the following *separated pieces property*:

For each disjunctive piece of the loop-condition, not overlapping any other pieces, the loop-body does not change the program variables in such a way that another piece becomes satisfied.

An example of a loop that does not satisfy this property is given in Listing 6.

```
1 while (i < 30 || (i > 29 && i < 100))
2   i++;
```

Listing 6. Loop in which the pieces are separate, but a jump is made from one piece to the other

5.2 Branching inside the loop body

The basic procedure finds correct LBFs 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 branches affect the bound in different ways. Such a case is shown in Listing 7.

```
1 while (i > 0)
2   if (i > 100) i -= 10;
3   else i -= 1;
```

Listing 7. Example where the basic method supplies an incorrect LBF. Therefore, branch-splitting is applied, yielding the untight, but correct LBF i .

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

With branch-splitting, we mean that we generate multiple new loops from the original, one for each possible branch. We then do the analysis for each of these branches. The LBF is then the maximum of all the inferred LBFs. Thanks to the WCCB property, we can easily find the LBF that always specifies the maximum, by supplying a set of values for the variables (say, all ones) to all the LBFs. For the example in Listing 7, this yields the LBF i .

A simple sub-class of loops with the WCCB property is given by if-statements breaking the loop execution, by a `return` or `break` or by throwing an exception. An example is shown in Listing 8.

```
1 for (int i = 0; i < a.length; i++)
2   if (a[i] < b[i]) return -1;
3   else if (a[i] > b[i]) return +1;
```

Listing 8. In this loop removing the if-statement yields a loop with the exact loop-bound function `a.length - i` that is the same as the worst-case bound of the original loop.

In general, in cases like this one may discard `return`-branches completely, since they do not yield an upper bound.

Another simple sub-class of loops with the WCCB property is given by loops containing if-statements, such that for *all* branches the values of the loop counter are the same. This happens when in the if-statement the values of variables, on which a given loop condition depends, are not changed. This is the case, for instance, when in the if-statement one changes the values in an array but not its length. Note, that when the loop parameters are changed in the same way in both the if and the else clauses then the if-statement may be transformed into an equivalent code fragment where it satisfies the condition above.

An example that does not satisfy the WCCB property is shown in Listing 9.

```
1 while (i > 0)
2   if (i % 2 == 0) i -= 3;
3   else i++;
```

Listing 9. Example that does not satisfy the WCCB property and is therefore not analysable using our method

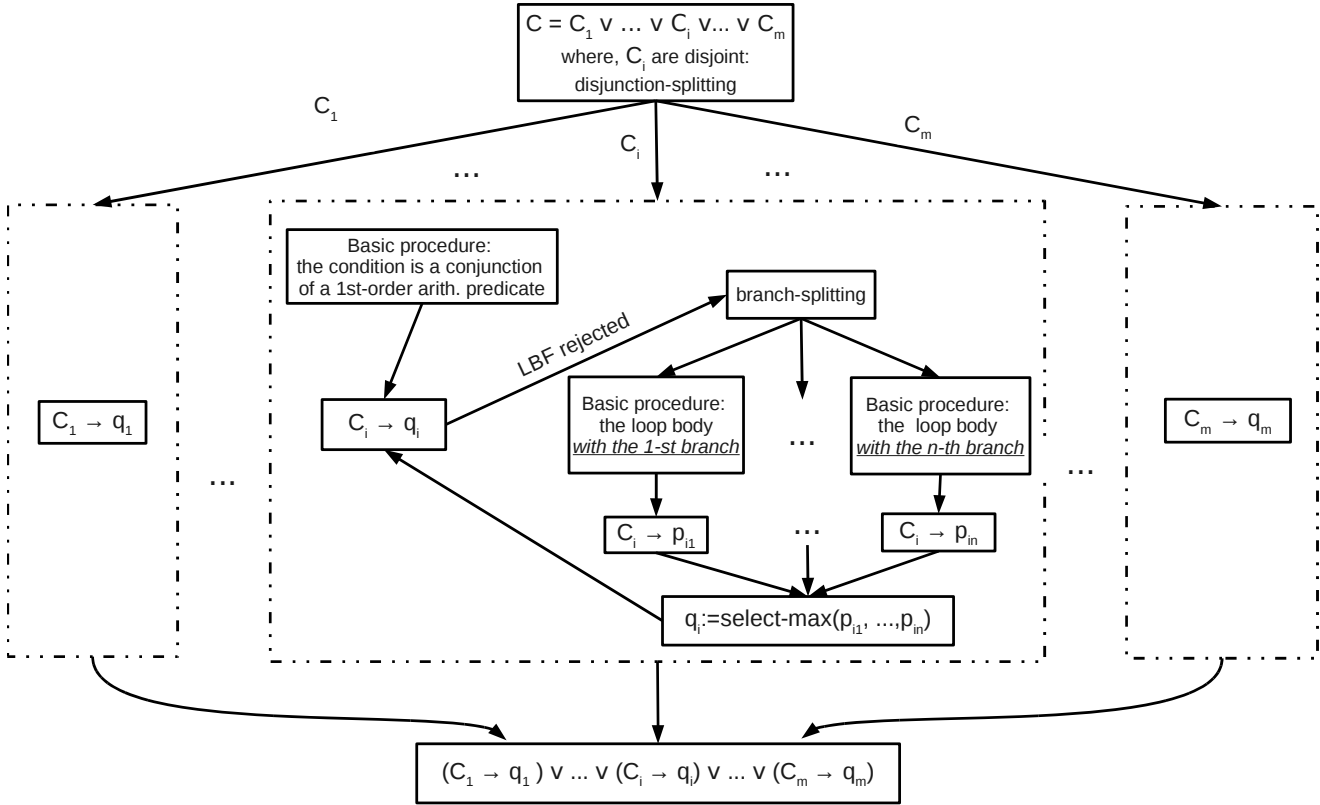


Figure 4. Overview of the inference procedure with extensions.

This example does not satisfy the property because executing the else-branch exclusively would yield an infinite bound. Not that it would also be impossible to formulate a correct decreases-clause for this loop.

An overview of the method with extensions is given in Figure 4. First we apply disjunction splitting, then we execute the basic method for each of the separate pieces. If this does not yet yield a correct LBF, then we apply branch-splitting and take the maximum of each of the LBFs of all the branches as LBF.

6. Evaluation

There are still some examples in the case studies that cannot yet be analysed using our method. At this point, we do not consider cases where the loop bound depends on:

- Fields of referenced objects
- Method invocations
- Booleans
- Different threads

A way of handling bounds that depend on references is described in [1], which we might incorporate in the future.

Furthermore, we require that the loops satisfy the separated pieces and worst-case computation branch properties. We believe that satisfying these properties is only a minor restriction, since rewriting loops to do so is usually fairly straight-forward. Such properties might even be including in the coding style requirements for safety-critical software.

For examples that can be handled by our method, it usually computes the *exact* LBF. An exception to this is when branch-splitting is applied. This means that compared to other methods, our method finds bounds that are equally tight, or tighter. Furthermore, other methods are unable to derive non-linear LBFs. This is discussed in more detail in the next section.

7. Related Work

Various other research results on bounding the number of loop iterations exist. However, most are concerned with concrete (numerical) bounds, instead of loop-bound *functions*. Also, most can only handle (tightly) cases where the bound depends linearly on program variables (we can handle the polynomial case). In a sense, our technique is more general than the methods discussed in this section. It may not be the most efficient method for simple loops, but it can be used to handle certain more complex cases. This makes it complementary to the other techniques discussed here.

Another common difference is that other approaches rely on handmade soundness proofs of their method, while we rely on a verification tool to ensure that the derived LBFs are correct.

In [8], pattern-matching on abstract syntax trees (ASTs) is used by Fulara et al to select one of several syntax-based schemes for generating decreases-clauses. If the AST matches a given pattern, then parameters from this pattern can be used to form a decreases-clause. The authors claim to cover 71% of all for-loops in a set of case studies. It is thinkable that their method is used in an implementation for the basic cases and our method is applied when no pattern matches.

Abstract interpretation, program slicing and invariant analysis are used by Ermedahl et al in [7] to infer numerical bounds for

C programs. The bounds meant here are integers representing the number of times a certain block of code is executed. The method can infer bounds for over 50% of the loops in a set of benchmarks.

A similar approach is taken by Lokuciejewski et al in [14], who combine abstract interpretation with polytope models to calculate numerical loop bounds for C programs. Both upper and lower bounds are calculated and the analysis is accelerated by using program slicing. Even though there are restrictive constraints on the loops that can be analysed, the authors claim that they can handle 99% of all for-loops in a set of benchmarks. Soundness or verification of the bounds are not discussed.

Abstract interpretation is also used in [6], in combination with flow analysis. Numerical bounds can be found for 84% of the loops in a benchmark suite. The method works on C programs.

Gulwani uses “off-the-shelf linear invariant generation tools” to compute symbolic loop bounds in [9]. The authors experiment with different counter instrumentation methods and a technique they named “control-flow refinement”. Loop-bound functions are presented as right-hand sides of the inequations in loop invariants. Inference of invariants is based on linear arithmetic, but some limited use of non-linear terms is possible as well. Given a particular program, the base arithmetic may be extended by a finite set of non-linear operators together with reasoning rules for them. The inference system, first, introduces a fresh variable for each non-linear operator, then deals with linear combinations of such variables (and usual arithmetic variables). The operators and the rules are chosen e.g. by a user, who knows which sort of invariants one can expect in the given code.

In a related article by the same author(s) [10], pattern-matching against known loop-iteration lemmas is used to establish bounds for C and C++ programs. This last method can find bounds for 93% of the loops in a significant Microsoft product.

In [4], Ben-Amram describes a method to derive *global* ranking functions, based on Size-Change Termination. Such a ranking function is required to decrease in each basic block of the program. He uses an abstraction called Monotonicity Constraints and represents them as graphs. Various algorithms are described that can be applied to these graphs to judge termination and construct ranking functions.

Hunt et al discuss the expression of manually conceived loop-bound functions in JML, their verification using KeY and the combination with data-flow analysis in [11]. This article is an important motivation for our work. What is “missing” in the method is the automated inference of loop-bound functions, which we supply.

In [2], Albert et al describe a system of generating and solving cost recurrence relations. These relations define functions that represent upper bounds on time or memory usage by a program. To solve a recurrence relation means to find a closed, i.e. a recursion-free, form of the corresponding function. Terms in the system represent *monotonic* real functions and, besides monotonically increasing polynomials, contain the exponent and the logarithmic functions.

8. Future Work

Here we discuss some areas where we will improve and extend our research and the prototype implementation in the future.

8.1 Improving test-node search

The first step in improving the implemented generation of test-nodes is to replace the linear programming solver that is used in the current prototype with a global optimisation library. This would enable us to handle loop conditions containing non-linear (in)equalities.

Next, the implemented search algorithm may be optimised, first of all, in its searching hyperplane-by-hyperplane part, probably

by memorising hyperplanes that have enough points for exactly a degree $d - i$ (see Section 3.1 for more detail).

When inferring the LBF for a loop with an increment > 1 the method does not always generate correct bound functions. In the case of linear polynomials inferred bound functions are correct but not optimal. In general, for a given loop, the connection between its nonlinear upper bound function, its incremental step and the interpolating polynomial needs to be studied. We have done the study for linear bounds, but the results have not been implemented yet.

Now we briefly discuss our observations about the connection between incremental steps and *linear loop-bound functions*. Strictly speaking, if $m > 1$ is an incremental step in a loop, then its worst-case bound is not exactly a polynomial but is of the form $\lceil \frac{p(\bar{z})}{m} \rceil$. Depending on the test nodes, our method finds a bound of the form $p_1(\bar{z}) + r$, where $0 \leq r \leq 1$. This can be explained by the fact that a graphical representation of $\lceil \frac{p(\bar{z})}{m} \rceil$ takes a step form, where the steps are of size m . For the interpolation to be correct, all test-nodes must be m apart, such that they are located at the same point w.r.t. the begin of a step interval. Furthermore, it is possible to infer the optimal *polynomial* bound (i.e. not ceiled) when the right test-nodes are chosen. This bound is $\frac{p(\bar{z}) + (m-1)}{m}$ for LBF-s of the aforementioned form. An example of such a loop is given in Listing 3. We want to choose such testing nodes ($\text{start}_i, \text{end}_i$) that $\text{end}_i - \text{start}_i \bmod m = 0$. In other words, test-nodes must be multiples of the step. Then the upper bound is $g(\bar{z}) + 1$ (or even $g(\bar{z}) + \frac{m-1}{m}$), where $g(\bar{z})$ is the data interpolating polynomial. We have been working on the procedure of generating test nodes for loops with steps even of more general form m/n , where $m > 1$. The task is formulated as follows: given a loop, one needs to generate the k -dimensional nodes, that:

- as earlier, are in NCA configuration,
- as earlier, satisfy the loop condition,
- are of the form $(\frac{m}{n}i_1, \dots, \frac{m}{n}i_k)$.

Instead of an original loop condition $P(x_1, \dots, x_k)$ consider the predicate $P'(x'_1, \dots, x'_k) = P(mx_1, \dots, mx_k)$. Using already existing part of the method, generate the intermediate list of integer nodes lying in NCA and satisfying this predicate:

$$[(x'_{11}, \dots, x'_{1k}), \dots, (x'_{N1}, \dots, x'_{Nk})]$$

The desirable list of test nodes is:

$$[(mx'_{11}, \dots, mx'_{1k}), \dots, (mx'_{N1}, \dots, mx'_{Nk})]$$

Indeed, the nodes in this list satisfy all three conditions above:

- they obviously lie in NCA, since it is just scaling of another NCA grid
- each of them satisfies the loop condition, because this holds:

$$P(mx'_{j1}, \dots, mx'_{jk}) = P'(x'_{j1}, \dots, x'_{jk})$$

- they are of the form $(\frac{m}{n}i_1, \dots, \frac{m}{n}i_k)$ because

$$(mx'_{j1}, \dots, mx'_{jk}) = (\frac{m}{n}nx'_{j1}, \dots, \frac{m}{n}nx'_{jk}),$$

$$\text{so } i_l = nx'_{jl}.$$

In the example of this section we interpolate a linear polynomial of two variables $p(\text{start}, \text{end}) = a \cdot \text{start} + b \cdot \text{end} + c$, which has three coefficients. The intermediate test nodes satisfy the predicate $4 \cdot \text{start} < 4 \cdot \text{end}$. They are, e.g. (0, 1), (0, 2), (1, 2). The corresponding test nodes are (0, 4) with the loop-iteration counter value 1, (0, 8), with the counter 2 and (4, 8) with the

counter 1. The corresponding interpolating polynomial is $\frac{\text{end-start}}{4}$ and a polynomial bound function is $\frac{\text{end-start}}{4} + \frac{3}{4}$.

8.2 Program slicing

As is done in [7] and [14], we intend to use program slicing to accelerate the analysis. This way, irrelevant parts of the code can safely be ignored. Slicing gives us exactly the variables on which the loop condition (i.e. the bound) depends. Since the complexity of our analysis is exponential in the number of variables in the polynomial, this is highly beneficial to the performance of the method.

8.3 Combination with data-flow analysis

The method could be combined with data-flow analysis in order to yield concrete numerical bounds. For example, when by flow analysis we can obtain an interval for each variable, then we can obtain a minimum and maximum on the number of loop iterations by searching for minima and maxima of the LBF within this interval.

8.4 Size and heap consumption analysis

The analysis should be combined with our previous research on size analysis. There is a mutual dependency between the two.

What size analysis may add to LBF inference is that a loop bound often depends on the size of a data structure (e.g. the length of a list). If we can bound this size, then we can bound the number of loop-iterations.

What LBF inference may add to size analysis is that often a data structure gets constructed by executing a loop. Think of iteratively adding elements to a list, for example when converting an array into a list object. If in that case we know the number of loop-iterations, we know the size of the constructed list.

Furthermore, objects may be constructed in a loop. This means that LBF inference is also crucial to bound heap-space usage, which is a future goal for the CHARTER project.

9. Conclusions

We have presented a way of computing arbitrary degree loop-bound functions. By expressing these functions in JML their correctness can be proved, which is very valuable in safety-critical systems. While various other methods for inferring loop-bounds exist, we are not familiar with any other works on generating non-linear loop-bound *functions* for Java. Moreover, the technique presented herein is largely complementary to other methods, since it is more general and can solve certain more complex cases, such as quadratic bounds.

Using a prototype implementation, loop-bound functions can be inferred for 66% of all loops in a set of case studies from actual safety-critical systems.

References

- [1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Dealing with numeric fields in termination analysis of java-like languages. In *FTJJP*, pages 77–87, 2008.
- [2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *SAS*, pages 221–237, 2008.
- [3] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
- [4] A. M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. In *CAV*, pages 109–123, 2009.
- [5] C. K. Chui and M.-J. Lai. Vandermonde determinants and lagrange interpolation in R^s . *Nonlinear and convex analysis*, pages 23–35, 1987.
- [6] 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. ISBN 978-0-7695-3349-0.
- [7] 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.
- [8] 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, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 978-3-642-11265-2.
- [9] 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. ISBN 978-3-642-02657-7.
- [10] 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. ISBN 978-1-60558-392-1.
- [11] 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. ISBN 1-59593-544-4.
- [12] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CD_X: a family of real-time java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems. JTRES 2009, Madrid, Spain, September 23-25, 2009*, pages 41–50. ACM, 2009.
- [13] 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.
- [14] 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. ISBN 978-0-7695-3576-0.
- [15] 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.
- [16] T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier, and M. Richard-Foy. Use of PERC Pico in the AIDA avionics platform. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 169–178. ACM, 2009.
- [17] 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.
- [18] 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.