Ranking Functions for Loops with Disjunctive Exit-Conditions

Rody Kersten¹ and Marko van Eekelen^{1,2}

Institute for Computing and Information Sciences (iCIS),
 Radboud University Nijmegen
 School for Computer Science, Open University of the Netherlands
 {R.Kersten, M.vanEekelen}@cs.ru.nl

Abstract. Finding ranking functions for the loops in a program is a prerequisite for proving its termination and analysing its resource usage. From its ranking function one easily derives a symbolic upper bound on the number of iterations of a loop. Such symbolic loop bounds can be used to derive concrete time and memory-usage bounds for complete programs.

This paper builds upon an earlier paper in which a polynomial interpolation based ranking function inference method is introduced for loops with exit conditions that are expressions in propositional logic over arithmetical (in)equalities.

We show that this earlier method is not applicable for certain loops: loops in which so-called *condition jumping* can occur. We define condition jumping and give an algorithm to detect it using symbolic execution and an SMT solver. We show how the earlier method can be adapted to be applicable also in the presence of condition jumping. As a result polynomial interpolation can be applied on a larger class of programs to infer polynomial ranking functions.

Keywords: Loop Bound, Ranking Function, Resource Analysis, Symbolic Execution, SMT solver

1 Introduction

Where software is used in safety-critical applications, it is important to derive and prove certain properties of this software, such as timing constraints and resource-usage bounds. It may be vital to an application that it runs within a fixed amount of time (e.g. when deploying an airbag) or memory (e.g. in embedded systems with limited capacity).

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

A 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++;
3 }</pre>
```

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

This paper builds upon an earlier paper in which a polynomial interpolation based ranking function inference method is introduced for loops with exit conditions that are expressions in propositional logic over arithmetical (in)equalities.

We show that this earlier method is not applicable for certain loops: loops in which so-called *condition jumping* can occur. The basic method, which considers only loops where the exit conditions are conjunctions over numerical (in)equalities, is described in Sect. 2. This method is extended to disjunctive exit conditions in Sect. 3, which requires *piecewise* ranking functions. At any point in execution, one part of a disjunctive exit condition may hold, then after execution of the loop body, another part of that condition may hold. This complicates our solution of splitting up the disjunctive parts of the condition. A formal definition of this *condition jumping* is given in Sect. 4, along with a method to detect it and an extension to the basic method which enables it to infer ranking functions also in the presence of condition jumping. Future work is discussed in Sect. 5 and related work in Sect. 6. The paper is concluded in Sect. 7.

2 Test Based Inference of Polynomial Ranking Functions for Loops using Polynomial Interpolation

In this section we describe what we will henceforth refer to as the *basic method*, first presented in [18]. In the basic method, we consider only loops for which the exit conditions are conjunctions over arithmetical (in)equalities:

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

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

The method works in the following steps:

- 1. Instrument the loop with a counter
- 2. Run test 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.

Polynomial interpolation theory is described in Sect. 2.1. In Sect. 2.2, the method is further clarified using an example. In Sect. 2.3, soundness is discussed.

2.1 Polynomial Interpolation

When the result of a polynomial function p is known for a certain set of test values W, the values of its coefficients a_0, \ldots, a_d can be derived (where d is the degree of the polynomial). Such a polynomial, which *interpolates* the test results, exists and is unique under certain conditions on the test data. These conditions are explored in polynomial interpolation theory [7].

In case p is a polynomial over a single variable z, this condition is well-known: W must contain d+1 pairwise different test-nodes. More specifically, if:

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

And for a set of d+1 pairwise different test-nodes z_0, \ldots, z_d we know the values of the polynomial function $p(z_0), \ldots, p(z_d)$, then we can find the unknown coefficients a_0, \ldots, a_d . These values of the polynomial function determine a system of linear equations with respect to the coefficients. This system of equations can be presented in matrix-form as follows:

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

There exists a unique solution to this system of equations and thus a unique polynomial interpolation in case the Vandermonde determinant of the matrix is non-zero. This is the case for pairwise different z_0, \ldots, z_d .

The condition under which the Vandermonde determinant is non-zero in the multivariate case is more complex. In [7], a series of *node configurations* under which a unique interpolation exists is presented. We use the simplest version: Node Configuration A, or NCA. It is only presented briefly here. It is described in more detail in [18], in which we first presented the basic method, and in [17], in which it is applied to test-data generation for size analysis of functional programs.

First, recall 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. For the two-dimensional case, NCA is defined as follows:

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

A typical instance of such a configuration is a 2-dimensional grid. For dimensions k > 2 the NCA is defined inductively on k:

 N_d^k nodes forming a set $W \subset \mathbb{R}^k$ lie in k-dimensional NCA if, for any $0 \le i \le d$, there is a (k-1)-dimensional hyperplane such that it contains N_{d-i}^{k-1} of the given nodes lying in (k-1)-dimensional NCA for the degree d-i and these nodes do not lie on any of the other hyperplanes.

Because we are applying the polynomial interpolation theory to the inference of ranking functions for loops, the test-nodes must also satisfy the exit condition of the analysed loop. An algorithm to find these test-nodes is described in [18]. It looks for test-nodes on a k-dimensional grid.

2.2 Quadratic Example

Consider the example in Listing 2. Its ranking function is the degree 2 polynomial $a \cdot b - c + 1$.

```
while (a > 0 && c <= b && c > 0) {
   if (c == b) {
        a = -;
        4      c = 0;
        }
        6      c++;
        7 }
```

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

The inference of a ranking function for the loop in Listing 2 is depicted in Fig. 1.

First, the loop is intrumented 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.

2.3 Soundness

The presented method infers a *hypothetical* ranking function. It is not sound by itself, but requires an external verifier.

We have chosen Java as source language in our implementation, therefore we use JML to express the ranking functions. However, there is no reason why the procedure would not work in other languages, such as C. When using C, the C-equivalent of JML, ACSL [4] might be used, which contains the same construct that we use to express ranking functions in JML.

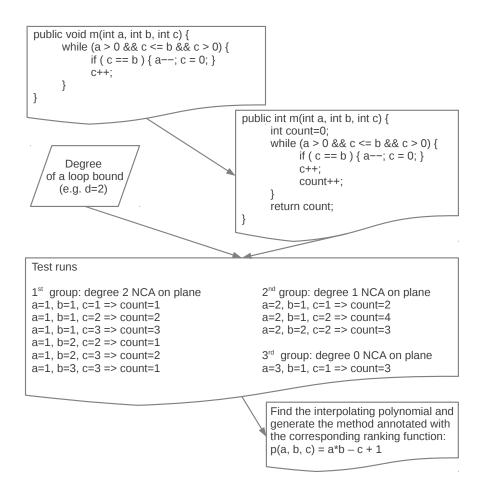


Fig. 1. Test-based inference method applied to the example from Listing 2.

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 that remains greater than or equal to 0, see the JML reference manual [15]. It therefore forms an upper-bound on the number of iterations of the loop. An example is shown in Listing 3.

```
1 //@ decreases i < 15 ? 15 - i : 0;
2 while (i < 15) {
3    i++;
4 }</pre>
```

Listing 3. The loop from Listing 1, annotated with its ranking function

When the loop condition does not hold, the loop iterates zero times. Therefore the shown annotation actually expresses the maximum of 15 - i and 0.

In general, a ranking function $RF(\bar{v})$ for a loop with condition b can be expressed as decreases b? $RF(\bar{v})$: 0.

Such JML annotations can be verified by a variety of tools. In our experiments we have used the KeY tool [5]. The procedure described in this paper should be used in conjunction with such a prover to provide soundness.

3 Piecewise Ranking Functions

In this section we extend the set of considered loops to those with as exit condition *any* propositional logical expression over arithmetical (in)equalities, thus now including disjunctions. We give a slightly more formal and thorough definition of our solution as in [18]. We will see that for those loops for which the exit condition contains disjunctions, the ranking function will become *piecewise*.

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

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

This is of course a trivial case. A more involved example of a loop for which a piecewise ranking function can be defined is shown in Listing 4.

```
while ((i>0 && i<20) || i>50) {
   if (i>50) i--;
        else i++;
        4 }
```

Listing 4. While loop with a piecewise ranking function.

It's ranking function is the following:

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

In this section we will show how the basic method is extended to treat loops with simple disjunctions in their exit conditions.

3.1 Extending the Basic Method

The exit condition of the loop in Listing 4 can easily be split up into three disjunctive parts: $i>0 \land i<20 \land \lnot(i>50),\ i>50 \land \lnot(i>0 \land i<20)$ and $(i>0 \land i<20) \land i>50$. The latter condition is not satisfiable by any i and is therefore removed.

Simplifying the remaining conditions leads to the following result: $Pieces = \{(i>0 \land i<20), (i>50)\}$, where Pieces defines the pieces of the piecewise polynomial ranking function. We can now execute the basic method for both of these pieces separately. This procedure leads to the ranking function in Eq. 2, as one would expect.

We will now formally define a generic method for inferring ranking functions for loops with disjunctive exit conditions, which we call *DNF-splitting*.

The first step is to transform the exit condition into disjunctive normal form (DNF), using the laws of distribution and DeMorgan's theorems. It thereafter thus has the form:

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

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

Let us shorten this to the following for readablity:

$$\bigvee_{i=1}^{n} b_i$$

So, each b_i represents a logical conjunction over numerical (in)equalities. We can now split up the exit condition by applying the function $DNFsplit :: C_d \to \{C_{nd}\}$, where C_d is the type representing conditions of the form described above and $\{C_{nd}\}$ is a collection of conditions in the form described in Sect. 2 (conjunctions over arithmetical (in)equalities).

$$DNFsplit(b_1 \lor \ldots \lor b_n) :=$$

$$\left\{ \bigwedge_{b_i \in BP} b_i \wedge \bigwedge_{b_j \in B_{rest}} \neg b_j \middle| BP \in \mathcal{P}(\{b_1, \dots, b_n\}) \backslash \emptyset \wedge B_{rest} = \{b_1, \dots, b_n\} \backslash BP \right\}$$

This transforms the condition $b_1 \vee ... \vee b_n$ into a set *Pieces* of $2^n - 1$ conjunctive conditions of type C_{nd} . This set may contain unsatisfiable conditions. The unsatisfiablity can be detected using an SMT solver and such conditions may safely be removed.

These conditions define the pieces of the piecewise polynomial ranking function. We can now apply the basic method separately for each of these pieces. If RF_p is the polynomial ranking function inferred for a piece $p \in Pieces$, then this yields the following piecewise ranking function:

$$\begin{cases}
RF_{p_1} & \text{if } p_1 \\
\dots & \text{if } \dots \\
RF_{p_m} & \text{if } p_m \\
0 & \text{else}
\end{cases}$$
(3)

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

3.2 Expressing Piecewise Ranking Functions in JML

By using the b? e_1 : e_2 notation for conditionals, we are able to express a piecewise polynomial decreases clause in JML. The piecewise ranking function 2 for the loop in Listing 4 is shown in Listing 5.

```
1 //@ decreases (i>0 && i<20) ? 20-i : ( i>50 ? i-50 : 0 );
2 while ((i>0 && i<20) || i>50) {
3   if (i>50) i--;
4   else i++;
5 }
```

Listing 5. Piecewise ranking function 2 expressed as a JML annotation.

In general, a ranking function of the form in Eq. 3 can be expressed as decreases p_1 ? RF_{p_1} : $(\ldots:(p_m?RF_{p_m}:0))$.

4 Condition Jumping

Building on the definitions from the previous section, we here define a complication that arises, which we call *condition jumping*. We also show how to detect its occurance and how to infer ranking functions even in the presence of condition jumping, which is our main contribution.

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

Listing 6. While loop with jumping between the disjunctive conditions.

Consider the loop in Listing 6. Naively, one could say that its ranking function is the following (the notation $\lceil n \rceil$ means n "ceiled" or rounded up):

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

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

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

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

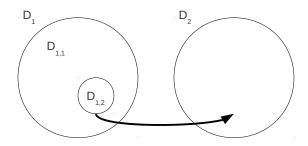


Fig. 2. Condition jumping

Figure 2 depicts condition jumping for a loop with exit condition $b_1 \vee b_2$. The total set of nodes satisfying b_1 is D_1 . The total set of nodes satisfying b_2 is D_2 . $D_{1,2}$ is the subset of D_1 for which jumping to b_2 occurs, $D_{1,1}$ is the subset for which jumping does not occur.

In the example, $D_1=[1,19],\, D_2=[23,\infty\rangle,\, D_{1,2}=\{n|n\in[1,19]\land n \text{ mod } 4=3\}$ and $D_{1,1}=D_1\backslash D_{1,2}.$

A method to detect condition jumping is described in Section 4.1. This method is then extended to detect all the nodes in $D_{1,2}$ in Section 4.2, in order to infer a correct piecewise ranking function.

4.1 Detection of Condition Jumping using Symbolic Execution and SMT Solvers

To detect condition jumping in the example in Listing 6, we will first use symbolic execution [14] to capture the relation between the values of the program variables pre and post execution of the loop body. We can then use this relation as input to an SMT solver and check if one part of the condition is true pre-execution of the body and another part is true post-execution.

We will name this pre/post execution relation for a variable v the $next_v$ function. The function $next_i::Int \to Int$ for the loop in from Listing 6 can be determined by symbolically executing the loop with value α_i for i. This results in the following symbolic post-execution value, which we will name ϕ_i :

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

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

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

An SMT-LIB script to detect jumping in the example from Listing 6 is given in Listing 7. The function $next_i :: Int \to Int$ from Equation 7 is defined on line 3. Then on line 4 we define the condition expressing that jumping occurs for this example and on line 5 we check satisfiability of this condition.

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

Listing 7. SMT-LIB script to detect jumping in the code of Listing 6.

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

For each $v_i \in LV$, we can define an associated function $next_{v_i} :: T_{v_1} \to \ldots \to T_{v_i} \to \ldots \to T_{v_n} \to T_{v_i}$, where T_{v_i} is the type of v_i and n = |LV|, which takes the values of all $v \in LV$ as the state and computes the value of v after a single execution of the loop body in that state.

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

Once these functions have been derived, the question whether jumping from b_1 to b_2 is possible can be answered by any SMT-LIB conforming SMT-solver³ by determining the satistiability of $b_1(v_1, \ldots, v_n) \wedge b_2(next_{v_1}(LV), \ldots, next_{v_n}(LV))$.

4.2 Generating Ranking Functions in the Presence of Condition Jumping

The SMT-LIB script in Listing 7 can be used to find a *model* for which jumping occurs by adding the expression (get-value (i)) between lines 5 and 6. A model is an instantiation of the variables for which the formula for which satisfiability is checked holds. In the SMT-LIB script from Listing 7, a model for i is 19.

Successively, the script in Listing 8 can now be used if there is any models other than i=19 for which jumping occurs. The answer of the SMT solver is that the combination of propositions in this script is unsatisfiable. Thus, i=19 is the only possible model.

³ For instance Z3, which can be used online at http://research.microsoft.com/en-us/um/redmond/projects/z3/

```
1 (set-logic QF_LIA)
2 (declare-fun i () Int)
3 (define-fun nexti ((x Int)) Int (ite (> x 22) (- x 1) (+ x 4)))
4 (assert (and (and (> i 0) (< i 20)) (> (nexti i) 22)))
5 (assert (distinct i 19))
6 (check-sat)
7 (get-value (i))
8 (exit)
```

Listing 8. SMT-LIB script to detect other models than i=19 for which jumping occurs in the code of Listing 6.

We can now see if there are any models from which the state i = 19 can be reached in a single iteration. The script in Listing 9 finds the model i = 15.

```
(set-logic QF_LIA)
(declare-fun i () Int)
(define-fun nexti ((x Int)) Int (ite (> x 22) (- x 1) (+ x 4)))
(define-fun nexti ((x Int)) Int (ite (> x 22) (- x 1) (+ x 4)))
(descent (and (and (> i 0) (< i 20)) (= (nexti i) 19)))
(descent (distinct i 19))
(discontant)
(descent (distinct i 19))
(discontant)
(discont
```

Listing 9. SMT-LIB script to detect models that can reach the state where i = 19.

Subsequently and similarly, we can search for other nodes that can reach the state i = 19 in a single step, or that can reach the state i = 15. By repeating these steps, we can find the set $D_{1,2} = \{3, 7, 11, 15, 19\}$. These are the models from which jumping can occur.

The term model is very similar to the term node. Both refer to an instantiation of program variables with a specific set of values. The term model is common in the area of SMT-solvers, while the term node is commonly used for test-points in the area of interpolation. The relation is as follows: $nodes \subseteq models$, because a set of test-nodes that is deemed suitable for interpolation is chosen from the total set of models satisfying a loop its exit condition. We will refer to a model with the vector \bar{v} .

In general, the method described in Section 4.1 can be extended to detect all models from which condition jumping can occur, by first finding all models that can jump directly from b_1 to b_2 and then recursively finding models that can reach a model from this first set. This can be done by implementing the following algorithm around an SMT-solver. In this algorithm, J is the set of models of which it is known that condition jumping occurs and Q is a queue of models. We assume a function $next :: M \to M$ (where M is the type of a model), which applies to each variable v_i in a model \bar{v} its corresponding $next_{v_i}$ function.

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

After execution, J contains exactly all elements from $D_{1,2}$. Since here a queue is used, this algorithm implements a breadth-first search. This can easily be adapted to a depth-first search by using a stack.

Now that we know $D_{1,2}$, we can split the condition b_1 into two: $b_1(\bar{v}) \wedge \bar{v} \in D_{1,2}$ and $b_1(\bar{v}) \wedge \bar{v} \notin D_{1,2}$. We can then apply the basic method to each of these disjunctive pieces.

4.3 Multi-Jumping

The algorithm described in the previous section detects jumping between two disjunctive parts of a condition. We need an umbrella algorithm to iteratively apply that pairwise algorithm in order to detect all jumping between all pieces.

For a condition b_1, \ldots, b_n , with respective sets D_1, \ldots, D_n of all the models satisfying them, the following algorithm can be used:

- 1. DNF-split into n conditions
- 2. For each i and j, $1 \le i < j \le n$, detect jumping from D_i to D_j . Build a list J of jumping pairs (D_x, D_y) for which condition jumping from D_x to D_y can occur.
- 3. If there are no more jumping pairs (D_x, D_y) for which D_x is unflagged, done! Else, goto 4.
- 4. Pop a jumping pair (D_x, D_y) off J, for which D_x is unflagged.
- 5. Apply the algorithm from Sect. 4.2 to find the set $D_{x,2}$ of all nodes in D_x from which jumping to D_y occurs and, dually, the set $D_{x,1}$ for which no jumping to D_y occurs. Replace any condition pair (D_x, D_z) in J by $(D_{x,1}, D_z)$. Add $(D_{x,2}, D_y)$ to J.
 - If $D_{x,1} = \emptyset$, flag $D_{x,2}$ as complete, goto 3.
 - Else, for any jumping pair (D_z, D_x) in J (i.e. for which jumping from D_z to D_x can occur), unflag D_z , detect jumping into $D_{x,1}$ and $D_{x,2}$ and update J accordingly. Goto 3.

5 Future Work

5.1 Implementation and Case Study

We have implemented the basic method and DNF-splitting for loops without condition jumping in a tool called ResAna. This tool can be downloaded at http://www.resourceanalysis.cs.ru.nl. We intend to implement the condition jumping detection method described in this paper as well, and adapt the implementation so that it can generate ranking functions in the presence of condition jumping.

When this implementation is done, the procedure can also be tested on a case study, to see which loops it can now analyse that were too complex before.

5.2 Use SMT Solver in Node-Search

In this paper, we use an SMT solver to detect condition jumping and to remove unsatisfiable conditions from the pieces list generated by DNF-splitting. The currently used algorithm to find test-nodes is described in [18], which searches for nodes in a quite extensive way. Node-search might be optimised by expressing NCA in SMT-LIB syntax and incrementally asking an SMT-solver to find a node, similar to the algorithm presented in Sect. 4.2.

6 Related Work

Various other research results on bounding the number of loop iterations are described in the literature. However, most apporaches generate concrete (numerical) bounds, 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) or that are constructed from monotonic subformulae. Several syntactical methods are discussed, that will be more efficient for simple cases, but less general. The procedure described in this paper can be seen as complementary for those methods. In case a syntactical method is not applicable to a certain loop, the more general method described here can be used.

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 [10], Fulara et al use pattern-matching on abstract syntax trees (ASTs) 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.

Abstract interpretation, program slicing and invariant analysis are used by Ermedahl et al in [9] 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 [16], 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 [8], 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.

In [6], 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.

Gulwani uses "off-the-shelf linear invariant generation tools" to compute symbolic loop bounds in [11]. The authors experiment with different counter instrumentation methods and a technique they named "control-flow refinement". Ranking functions are presented as 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 [12], Gulwani and Zuleger present another method to derive ranking functions. First, a transition system representing the program is built. Then, patterns are matched against this transition system. This leads to disjunctive ranking functions for all transitions, which can successively be combined into a global ranking function. This method reputedly achieves better results for nested loops and is able to compute ranking functions for 76% of the loops in a .Net base-class library. They use an SMT solver to verify ranking functions over loop-free program fragments.

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 [13]. This article is an important motivation for our work. What is "missing" in the method is the automated inference of ranking functions, which we supply.

In [1], 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. Their method is implemented in the COSTA system, which is described in [2]. In [3], an approach that is similar to ours in ResAna [18] 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.

7 Conclusions

We have shown that the previously published polynomial interpolation based ranking function inference method is not applicable for certain loops: loops in which so-called *condition jumping* can occur. We have given a definition of condition jumping and presented an algorithm to detect it using symbolic execution and an SMT solver. Also, the earlier method has been adapted to be applicable also in the presence of condition jumping. This extension makes the method more powerful, in the sense that it can now infer symbolic loop bounds also for loops in which this condition jumping occurs.

Ranking functions for loops can be combined with ranking functions for the other statements and constructions in a program to create a global ranking function. The existence of a global ranking function proves termination of the program.

Also, ranking functions for loops are vital to an analysis of resource-usage. For instance, if a loop with ranking function $RF(\bar{v})$ contains the statement new ObjOnHeap, then we know that the execution of that loop will consume $RF(\bar{v}) \cdot size(ObjOnHeap)$ bytes of heap-space. Other examples of resources that are often consumed proportionally to a symbolic loop bound include time, network traffic and energy.

It is therefore crucial for resource and termination analysis that (precise) ranking functions for *all* loops in a program can be defined, not just the simple linear cases or those that match a user-defined pattern. The basic polynomial interpolation method already formed a more general solution and with the extension presented in this paper, we are one step further in achieving this goal.

References

- Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: SAS. pp. 221–237 (2008)
- Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F., Bonsangue, M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, Lecture Notes in Computer Science, vol. 5382, pp. 113–132. Springer Berlin / Heidelberg (2008)
- 3. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: Verified resource guarantees using COSTA and KeY. In: Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation. pp. 73–76. PEPM '11, ACM, New York, NY, USA (2011)

- Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.5 (2010), http://frama-c.com/acsl.html
- Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS 4334, Springer-Verlag (2007)
- Ben-Amram, A.M.: Size-change termination, monotonicity constraints and ranking functions. In: CAV. pp. 109–123 (2009)
- 7. Chui, C.K., Lai, M.J.: Vandermonde determinants and lagrange interpolation in \mathbb{R}^s . Nonlinear and convex analysis pp. 23–35 (1987)
- 8. De Michiel, M., Bonenfant, A., Cassé, H., Sainrat, P.: 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. pp. 161–166. IEEE Computer Society, Washington, DC, USA (2008)
- 9. Ermedahl, A., Sandberg, C., Gustafsson, J., Bygde, S., Lisper, B.: Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In: Rochange, C. (ed.) 7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany (2007)
- Fulara, J., Jakubczyk, K.: Practically applicable formal methods. In: SOFSEM '10: Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science. pp. 407–418. Springer-Verlag, Berlin, Heidelberg (2010)
- 11. Gulwani, S.: SPEED: Symbolic complexity bound analysis. In: CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification. pp. 51–62. Springer-Verlag, Berlin, Heidelberg (2009)
- 12. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation. pp. 292–304. PLDI '10, ACM, New York, NY, USA (2010)
- 13. Hunt, J.J., Siebert, F.B., Schmitt, P.H., Tonin, I.: 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. pp. 162–169. ACM, New York, NY, USA (2006)
- King, J.C.: Symbolic execution and program testing. Commun. ACM 19, 385–394 (July 1976)
- Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual. Draft Revision 1.200 (Feb 2007)
- 16. Lokuciejewski, P., Cordes, D., Falk, H., Marwedel, P.: 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. pp. 136–146. IEEE Computer Society, Washington, DC, USA (2009)
- 17. Shkaravska, O., van Eekelen, M., van Kesteren, R.: Polynomial size analysis of first-order shapely functions. Logic in Computer Science 2:10(5) (2009)
- 18. Shkaravska, O., Kersten, R., Van Eekelen, M.: Test-based inference of polynomial loop-bound functions. In: Krall, A., Mössenböck, H. (eds.) PPPJ'10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java. pp. 99–108. ACM Digital Proceedings Series (2010)