Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs

Christophe Alias[1], Alain Darte[1], Paul Feautrier[1], and Laure Gonnord[2]

[1] Compsys team, LIP, Lyon, France
UMR 5668 CNRS—ENS Lyon—UCB Lyon—Inria
{Firstname.Lastname}@ens-lyon.fr
[2] LIFL - UMR CNRS/USTL 8022, INRIA Lille - Nord Europe
40 avenue Halley, 59650 Villeneuve d'Ascq, France
Laure.Gonnord@lifl.fr

Abstract. Proving the termination of a flowchart program can be done by exhibiting a ranking function, i.e., a function from the program states to a well-founded set, which strictly decreases at each program step. A standard method to automatically generate such a function is to compute invariants for each program point and to search for a ranking in a restricted class of functions that can be handled with linear programming techniques. Previous algorithms based on affine rankings either are applicable only to simple loops (i.e., single-node flowcharts) and rely on enumeration, or are not complete in the sense that they are not guaranteed to find a ranking in the class of functions they consider, if one exists. Our first contribution is to propose an efficient algorithm to compute ranking functions: It can handle flowcharts of arbitrary structure, the class of candidate rankings it explores is larger, and our method, although greedy, is provably complete. Our second contribution is to show how to use the ranking functions we generate to get upper bounds for the computational complexity (number of transitions) of the source program. This estimate is a polynomial, which means that we can handle programs with more than linear complexity. We applied the method on a collection of test cases from the literature. We also show the links and differences with previous techniques based on the insertion of counters.

1 Introduction and Motivation

The problem of proving program correctness has been with us since the early days of Computer Science. In a seminal paper [20], R. W. Floyd proposed what has become one of the standard approaches: affix assertions to each program point and prove that they are consequences of the assertions of its predecessors in the program control graph. The assertions at the entry point of the program are its *preconditions*, the assertions at loop entry points are *invariants*, while the assertions at its exit point must entail correctness, according to some set of requirements. Constructing the required set of assertions is a tedious and error-prone task. The automatic construction of invariants has been proved to be intractable in the general case [6]. However, partial or conservative solutions can be obtained by abstract interpretation methods [14].

At the same time, it was soon realized that this method proves only partial correctness, i.e., that the program gives the correct result if and when it terminates. To prove

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 117–133, 2010.

termination, one needs a variant or ranking function (a W-function in Floyd's terminology), i.e., a function from the states of the program to some well-founded set, which strictly decreases at each program step. Of course, designing an algorithm for building ranking functions in all cases is not possible since it would give a solution to the undecidable halting problem. However, this does not preclude the existence of partial solutions, which, e.g., handle only programs (or approximated models) of a restricted shape, or look for rankings in a restricted class of functions. Our first contribution is to generalize previous work for generating ranking functions. We design an algorithm with the following features:

- It can handle flowcharts of arbitrary structure.
- The class of rankings we consider is much larger: in the global ranking function we generate, each program point can have its own multi-dimensional affine expression.
- Our algorithm is based on a greedy mechanism. Nevertheless, our technique is provably complete, even for our larger class of ranking functions.

There are many variations on the above theme. For instance, as in [27], one may select a set of *cutpoints*, with the property that their removal makes the flowchart acyclic. It is then enough to exhibit a function, non increasing everywhere, that decreases and is well-founded at each cutpoint. One may even proceed each flowchart cycle at a time.

Our second contribution is to show that the global ranking functions we generate can be used to give upper bounds on the worst-case computational complexity (WCCC) of the program execution, i.e., the number of transitions that can be made in an execution trace. Obviously, if a program does not terminate, its WCCC is infinite. If the program terminates and a one-dimensional ranking function exists, its value at program start is an upper bound on the number of steps before termination since it decreases at least by one at each program step. The situation is more complicated in the case of multi-dimensional ranking functions but we show how the WCCC can be computed thanks to counting techniques in polyhedra. Furthermore, our ranking algorithm has an additional important feature: It generates a multi-dimensional affine ranking function whose dimension is minimal. This is important to get an accurate upper bound on the WCCC of the flowchart program. To the best of our knowledge, our technique is the first one that uses ranking functions to compute upper bounds on the number of iterations of arbitrary loops (a particular case of the WCCC).

The rest of the paper is organized as follows. Section 2 gives some basic notations and concepts: the program abstraction we use (*integer interpreted automata*) and the class of ranking functions we consider. Section 3 presents our method for constructing multi-dimensional affine ranking functions and states its completeness. Section 4 explains how we infer the computational complexity of the source program. Section 5 reports on our implementation through a collection of benchmarks from the literature. Section 6 describes other approaches to the termination problem and WCCC evaluation. We then conclude pointing to some unsolved problems and outlining future work.

2 Notations and Definitions

We write matrices with capital letters (as A) and column vectors in boldface (as $\boldsymbol{x}$). If $\boldsymbol{x}$ has dimension d, its components are denoted $\boldsymbol{x}[i]$, with $0 \leq i < d$. Thus, its i-th component is $\boldsymbol{x}[i-1]$. Sets are represented with calligraphic letters such as $\mathcal{W}$, $\mathcal{K}$, etc.

2.1 Integer Interpreted Automata

In the tradition of most previous work on program termination and static analysis, we first transform the program to be analyzed into an abstraction: the associated *integer interpreted automaton*. This is similar to the flowcharts used a long time ago to express programs (see, e.g., Manna's book [27]) until the advent of structured programming. In fact, when one looks at real-life programs, many deviations from the strict structured model can occur, including premature loop termination, exceptions, and even the occasional `goto`. Reasoning with flowcharts abstracts the details of the syntax and semantics of the source language, which can be dealt with by an appropriate preprocessor.

In our work, a program is represented by an *affine (integer) interpreted automaton* $(\mathcal{K}, n, k_{init}, \mathcal{T})$ defined by:

- a finite set $\mathcal{K}$ of *control points*;
- n integer variables represented by a vector $\boldsymbol{x}$ of size n;
- an initial control point $k_{init} \in \mathcal{K}$;
- a finite set $\mathcal{T}$ of 4-tuples (k, g, a, k'), called *transitions*, where $k \in \mathcal{K}$ (resp. $k' \in \mathcal{K}$) is the source (resp. target) control point, $g : \mathbb{Z}^n \mapsto \mathbb{B} = \{\text{true}, \text{false}\}$, the *guard*, is a logical formula expressed with affine inequalities $G\boldsymbol{x} + \boldsymbol{g} \geq 0$, and $a : \mathbb{Z}^n \mapsto \mathbb{Z}^n$, the *action*, assigns, to each variable valuation $\boldsymbol{x}$, a vector $\boldsymbol{x}'$ of size n, expressed by an affine expression $\boldsymbol{x}' = A\boldsymbol{x} + \boldsymbol{a}$. Here, G and A are matrices, $\boldsymbol{g}$ and $\boldsymbol{a}$ are vectors.

To represent non-determinism or to approximate non-affine or non-analyzable assignments in the program, we may have to assign the value "?", representing an arbitrary integer, to a variable, but we will not elaborate on this point. This is equivalent to deal with affine relations between $\boldsymbol{x}$ and $\boldsymbol{x}'$ instead of functions, see [1] for details.

Semantics. The set of states is $\mathcal{K} \times \mathbb{Z}^n$. A *trace* from $(k_0, \boldsymbol{x}_0)$ to $(k, \boldsymbol{x})$ is a sequence $(k_0, \boldsymbol{x_0}), (k_1, \boldsymbol{x}_1), \ldots, (k_p, \boldsymbol{x}_p)$ such that $k_p = k$, $\boldsymbol{x}_p = \boldsymbol{x}$ and for each i, $0 \leq i < p$, there exists in $\mathcal{T}$ a transition (k_i, g_i, a_i, k_{i+1}) such that $g_i(\boldsymbol{x}_i) = \text{true}$ and $\boldsymbol{x}_{i+1} = a_i(\boldsymbol{x}_i)$. Given an initial valuation $\boldsymbol{v}$, a state $(k, \boldsymbol{x})$ is *reachable from* $\boldsymbol{v}$ iff (if and only if) there is a trace from $(k_{init}, \boldsymbol{v})$ to $(k, \boldsymbol{x})$. A state $(k, \boldsymbol{x})$ is *reachable* if there exists $\boldsymbol{v} \in \mathbb{Z}^n$ such that $(k, \boldsymbol{x})$ is reachable from $\boldsymbol{v}$. The set of reachable states is denoted by $\mathcal{R}$.

Invariants. The guard g in a transition $t = (k, g, a, k')$ gives a necessary condition on variables $\boldsymbol{x}$ to traverse the transition t and to apply its corresponding action a. To get the exact valuations $\boldsymbol{x}$ of variables for which the action a can be performed, one would need to take into account the initial valuations and the successive conditions that led to the control point k. We denote by $\mathcal{R}_k$ the set of possible valuations $\boldsymbol{x}$ of variables when the control is in k:

$$\mathcal{R}_k = \{\boldsymbol{x} \in \mathbb{Z}^n \mid (k, \boldsymbol{x}) \in \mathcal{R}\}.$$

Then, there exists a trace containing the transition (k, g, a, k') iff $\boldsymbol{x} \in \mathcal{R}_k$ and $g(\boldsymbol{x})$ is true. Note that $\mathcal{R}_k$ does not depend on any initial valuation. More precisely, it is the union, for all initial valuations $\boldsymbol{v}$, of the set of vectors $\boldsymbol{x}$ such that $(k, \boldsymbol{x})$ is reachable from $\boldsymbol{v}$.

In practice, it is difficult to determine the set $\mathcal{R}_k$ exactly but it is possible to give over-approximations, thanks to the notion of *invariants*. An invariant on a control point k is a formula $\phi_k(\boldsymbol{x})$ that is true for all reachable states $(k, \boldsymbol{x})$. It is *affine* if it is the conjunction

of a finite number of affine conditions on program variables. The set $\mathcal{R}_k$ is then over-approximated by the integer points within a polyhedron $\mathcal{P}_k$. To compute invariants, we rely on standard *abstract interpretation* techniques, widely studied since the seminal paper of Cousot and Halbwachs [14]. These sets $\mathcal{P}_k$ represent all the information on the values of variables that can be deduced from the program by state-of-the-art analysis techniques. Unlike [8,24] where the construction of invariants is coupled with the termination proof or evaluation of iteration bounds, the invariants $\mathcal{P}_k$ are pre-computed and are the inputs of the techniques developed in the next sections.

2.2 Termination and Ranking Functions

Invariants can only prove partial correctness of a program. The standard technique for proving termination is to consider ranking functions to well-founded sets. A well-founded set $\mathcal{W}$ is a set with a (total or partial) order $\preceq$ (we write $a \prec b$ if $a \preceq b$ and $a \neq b$) such that there is no infinite descending chain, i.e., no infinite sequence $(x_i)_{i\in\mathbb{N}}$ with $x_i \in \mathcal{W}$ and $x_{i+1} \prec x_i$ for all $i \in \mathbb{N}$.

Definition 1. *A* ranking *is a function $\rho : \mathcal{K} \times \mathbb{Z}^n \to \mathcal{W}$, from the automaton states to a well-founded set $(\mathcal{W}, \preceq)$, whose values decrease at each transition $t = (k, g, a, k')$:*

$$\boldsymbol{x} \in \mathcal{R}_k \wedge g(\boldsymbol{x}) = \mathit{true} \wedge \boldsymbol{x}' = a(\boldsymbol{x}) \Rightarrow \rho(k', \boldsymbol{x}') \prec \rho(k, \boldsymbol{x}) \tag{1}$$

It is said affine *if it is affine in the second parameter (the variables).*

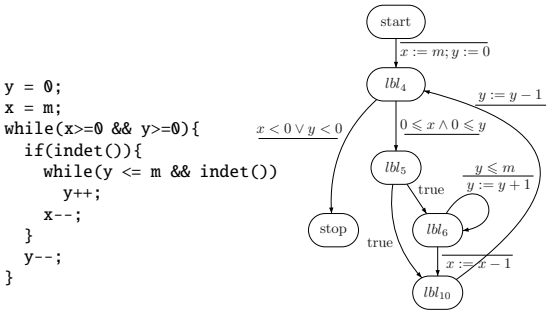
Definition 2. *A ranking function is* one-dimensional *if its co-domain is $(\mathbb{N}, \leq)$. It is* k-dimensional *(or multi-dimensional of dimension k) if its co-domain is $(\mathbb{N}^k, \preceq_k)$, where the order $\preceq_k$ is the standard lexicographic order on integer vectors.*

Obviously, the existence of a ranking function implies program termination for any valuation $\boldsymbol{v}$ at the initial control point k_{init}. A well-known property is that an integer interpreted automaton terminates for any initial valuation if and only if it has a ranking function. Furthermore, if it terminates and has bounded non-determinism, there is a one-dimensional ranking function, which is not necessarily affine.

2.3 Illustrating Example

An example program is given in Fig. 1, with its corresponding automaton. The control points are labelled for convenience, and transitions are depicted with arrows indexed by $\frac{g}{a}$ (g is omitted when g = true). State names are assigned arbitrarily by our parser.

The C code features two nested loops, which do not fit into the structured programming model, since the inner counter, `y`, is modified in the outer loop. The `indet` function abstracts non-determinism or an intractable test. The outcome of non-determinism is that, in the corresponding automaton, both transitions out of state lbl_5 have a true guard. The right of Fig. 1 successively gives, assuming $m > 0$, the invariants as found by Aspic (an abstract-interpretation based invariant generator, see Section 5), followed by the bidimensional rankings and the corresponding WCCC computed by Rank, our tool. The reader may care to check that these rankings are positive and lexicographically decrease along each transition. For instance, the first component of the ranking function decreases from $2x + 3$ at lbl_5 to $2x + 2$ at lbl_6, then $2x + 3$ at lbl_{10}, but since x is changed to $x - 1$ by the corresponding transition, the ranking has really decreased.



lbl_4	$m \geq x > 0, m \geq y > 0$
lbl_5	$m \geq x \geq 0, m \geq y \geq 0$
lbl_6	$m \geq x \geq 0, m + 1 \geq y \geq 0$
lbl_{10}	$m \geq x \geq -1, m + 1 \geq y \geq 0$ $2m \geq x + y$
start	$2m + 4$
lbl_4	$(2x + 3, 3y + 3)$
lbl_5	$(2x + 3, 3y + 2)$
lbl_6	$(2x + 2, m - y + 1)$
lbl_{10}	$(2x + 3, 3y + 1)$
wccc	$5 + 7m + 4m^2$

Fig. 1. Illustrating example

3 Computing Affine Ranking Functions

This section gives an algorithm to build a multi-dimensional affine ranking function, i.e., a ranking function $\rho : \mathcal{K} \times \mathbb{Z}^n \to \mathbb{N}^d$, affine for the second parameter. The integer d is the dimension of the ranking. Considering ranking functions with $d > 1$ is mandatory to be able to prove the termination of programs that induce a number of transitions, i.e., a trace length, more than linear in the program parameters. Furthermore, when a d-dimensional ranking exists, the number of transitions can be bounded by a polynomial, derived from the ranking, with a simpler method than by manipulating directly polynomials of degree d. Considering rankings with a different affine function for each control point also extends the set of programs whose termination can be determined, compared for example to the technique of [13] (see more details in Section 3.2).

3.1 A Greedy Polynomial-Time Procedure

As explained in Section 2.1, in practice, the exact sets $\mathcal{R}_k$ are not necessarily available. They are over-approximated by invariants $\mathcal{P}_k$, with $\mathcal{R}_k \subseteq \mathcal{P}_k$, which are, in our case, described by polyhedra. The conditions that a ranking function must satisfy are then related to these invariants and not to the exact sets of reachable states.

A ranking function ρ of dimension d needs to satisfy two properties. First, as ρ has co-domain $\mathbb{N}^d$, it should assign a nonnegative integer vector to each relevant state:

$$\boldsymbol{x} \in \mathcal{P}_k \Rightarrow \rho(k, \boldsymbol{x}) \geq \mathbf{0} \text{ (component-wise)} \tag{2}$$

Second, it should decrease on transitions. Let $\mathcal{Q}_t$ be the polyhedron described by the constraints of a transition $t = (k, g, a, k')$, i.e., $\boldsymbol{x} \in \mathcal{P}_k$, $g(\boldsymbol{x})$ is true, and $\boldsymbol{x}' = a(\boldsymbol{x})$, which can be built from matrices A and G, and vectors $\boldsymbol{a}$ and $\boldsymbol{g}$ (see Section 2.1). For an automaton whose actions are general affine *relations*, $\mathcal{Q}_t$ is directly given by the action definitions. With $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}') = \rho(k, \boldsymbol{x}) - \rho(k', \boldsymbol{x}')$, Inequality (1) then becomes:

$$(\boldsymbol{x}, \boldsymbol{x}') \in \mathcal{Q}_t \Rightarrow \Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}') >_d \mathbf{0} \tag{3}$$

which means $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}') \neq \mathbf{0}$ and its first nonzero component is positive. It this component is the i-th, the *level* of $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}')$ is i. A transition t is said to be (fully) *satisfied by the i-th component* of ρ (or *at dimension i*) if the maximal level of all $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}')$ is i.

To build a ranking ρ, the difficulty is to decide, for each transition t and for each pair $(\boldsymbol{x}, \boldsymbol{x}') \in \mathcal{Q}_t$, what will be the level of $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}')$ and by which component of ρ the transition t will be satisfied. A potentially exponential search, as in [8], should be avoided. To address this issue, our algorithm uses the same greedy mechanism as in [25,19,13]. The components of ρ are functions from $\mathcal{K} \times \mathbb{Z}^n$ to $\mathbb{N}$. We build them, one after the other, from the first one to the last one. For a component σ of ρ and a transition t not yet satisfied by one of the previous components of ρ, we consider the constraint:

$$(\boldsymbol{x}, \boldsymbol{x}') \in \mathcal{Q}_t \Rightarrow \Delta_t(\sigma, \boldsymbol{x}, \boldsymbol{x}') \geq \varepsilon_t \text{ with } 0 \leq \varepsilon_t \leq 1. \tag{4}$$

and we select a ranking such that as many transitions as possible have $\varepsilon_t = 1$, i.e., are now satisfied. Surprisingly, despite this *greedy* approach, our technique is provably complete (see Theorem 1), which means that if a multi-dimensional affine ranking exists, our algorithm finds one. Our algorithm can then be summarized as follows:

1: $i = 0$; $T = \mathcal{T}$; ▷ Initialize T to the set of all transitions
2: **while** T is not empty **do**
3: Find a 1D affine function σ and values ε_t such that all inequalities (2) and (4) are satisfied and as many ε_t as possible are equal to 1; ▷ This means maximizing $\sum_{t \in T} \varepsilon_t$
4: Let $\rho_i = \sigma$; $i = i + 1$; ▷ σ defines the i-th component of ρ
5: If no transition t with $\varepsilon_t = 1$, **return** false ▷ No multi-dimensional affine ranking.
6: Remove from T all transitions t such that $\varepsilon_t = 1$; ▷ The transitions have level i
7: **end while**;
8: $d = i$; **return** true; ▷ There is a d-dimensional ranking

For Line 3, any solution σ leading to $\varepsilon_t > 0$ can be multiplied by a suitable positive constant to get a solution with $\varepsilon_t = 1$. Thus, for any solution maximizing $\sum_{t \in T} \varepsilon_t$, a transition t has either $\varepsilon_t = 0$ or $\varepsilon_t = 1$. At each iteration of the while loop, σ is used as a new component of the ranking ρ (Line 4). By construction, ρ is strictly decreasing at this level for all transitions t with $\varepsilon_t = 1$. No need to consider them any longer, which means that they are removed for building subsequent components (Line 6). If no transition is removed, no ranking function is derived and the automaton may not terminate.

To find a suitable function σ at Line 3, we use linear programming. The set of inequalities that we need to solve are Inequalities (2) (with σ instead of ρ) and (4). The standard method (used in [19,28,8]) is to rely on the affine form of Farkas lemma [30]:

Lemma 1 (Farkas lemma, affine form). *An affine form $\phi : \mathbb{R}^n \to \mathbb{R}$ with $\phi(\boldsymbol{x}) = \boldsymbol{c}.\boldsymbol{x} + c_0$ is nonnegative everywhere in a non-empty polyhedron $\{\boldsymbol{x} \mid A\boldsymbol{x} + \boldsymbol{a} \geq \boldsymbol{0}\}$ iff:*

$$\exists \boldsymbol{\lambda} \in (\mathbb{R}^+)^n, \lambda_0 \in \mathbb{R}^+ \text{ such that } \phi(\boldsymbol{x}) \equiv \boldsymbol{\lambda}.(A\boldsymbol{x} + \boldsymbol{a}) + \lambda_0$$

The notation $\equiv$ is a formal equality, which means that $\boldsymbol{x}$ can be eliminated and coefficients identified. In other words:

$$\exists \boldsymbol{\lambda} \in (\mathbb{R}^+)^n, \lambda_0 \in \mathbb{R}^+ \text{ such that } \boldsymbol{c} = \boldsymbol{\lambda}.A \text{ and } c_0 = \boldsymbol{\lambda}.\boldsymbol{a} + \lambda_0$$

We can now apply the affine form of Farkas lemma to Inequalities (2) (with σ instead of ρ) and (4). With $\mathcal{P}_k = \{\boldsymbol{x} \mid P_k\boldsymbol{x} + \boldsymbol{p_k} \geq \boldsymbol{0}\}$, we transform Inequality (2) into:

$$\exists \boldsymbol{\lambda_k} \in (\mathbb{R}^+)^n, \lambda_k^0 \in \mathbb{R}^+ \text{ such that } \sigma(k, \boldsymbol{x}) \equiv \boldsymbol{\lambda_k}.(P_k\boldsymbol{x} + \boldsymbol{p_k}) + \lambda_k^0 \tag{5}$$

Similarly, with $Q_t = \{\boldsymbol{y} = (\boldsymbol{x}, \boldsymbol{x}') \mid Q_t\boldsymbol{y} + \boldsymbol{q_t} \geq \boldsymbol{0}\}$, we transform Inequality (4) into:

$$\exists \boldsymbol{\mu_t} \in (\mathbb{R}^+)^n, \mu_t^0 \in \mathbb{R}^+ \text{ s.t. } \Delta_t(\sigma, \boldsymbol{x}, \boldsymbol{x}') - 1 \equiv \boldsymbol{\mu_t}.(Q_t\boldsymbol{y} + \boldsymbol{q_t}) + \mu_t^0 \tag{6}$$

A substitution of (5) in (6) and an identification on each dimension of $\boldsymbol{y}$ leads to a set of linear inequalities. Considering all inequalities obtained for all transitions $t \in T$ and maximizing $\sum_{t\in T} \varepsilon_t$ (Line 3 of the algorithm) leads to the desired function σ.

Note: As we use linear programming, but not integer linear programming, we may end up with a function σ with rational components. However, we can always multiply it by a suitable integer to get a ranking function with integer values.

Example of Section 2.3 (Cont'd). Write $\sigma_k(x, y) = a_k x + b_k y + c_k$ the 1st component of the ranking. Consider any transition, e.g., $lbl_4 \to lbl_5$. The non-increasing constraint gives $(a_4 - a_5)x + (b_4 - b_5)y + c_4 - c_5 \geq 0$. Letting $x = 0$ and $y = m$, and noticing that m can be arbitrarily large, gives $b_4 \geq b_5$. The same technique applied to all transitions of a cycle shows that all b_k (same for all a_k) of a strongly connected component are equal: let b this value. For the self-loop on lbl_6, $\sigma_6(x, y) \geq \sigma_6(x, y+1)$ implies $b \leq 0$. The cycle $lbl_4 \to lbl_5 \to lbl_{10} \to lbl_4$ implies $\sigma_4(x, y) \geq \sigma_4(x, y-1)$, thus $b \geq 0$. Hence, these two cycles cannot be satisfied at the first dimension. However, the transitions $lbl_5 \to lbl_6$ and $lbl_6 \to lbl_{10}$ can be satisfied, disconnecting the two cycles and allowing them to be satisfied separately by the 2nd component of ρ. Here, we have deliberately simplified the problem by ignoring the positivity constraints and using qualitative reasoning for analyzing the descent constraints. In our tool, linear programming replaces intuition.

3.2 Completeness

Since non-terminating programs exist, there is no hope of proving that a ranking function always exists. Moreover, there are terminating affine interpreted automata with no multi-dimensional affine ranking. Thus, all we can prove is that, if a multi-dimensional affine ranking exists, our algorithm finds one, i.e., it is *complete* for the class of multi-dimensional affine rankings. Also, as the sets $\mathcal{R}_k$ are over-approximated by the invariants $\mathcal{P}_k$, completeness has to be understood with respect to these invariants, which means that if the algorithm fails when an affine ranking exists, it is because invariants are not accurate enough. In this section, we just sketch the completeness proof. The proof itself, quite long and technical, can be found in the long version of this paper [2].

Theorem 1. *If an affine interpreted automaton, with associated invariants, has a multi-dimensional affine ranking function, then the algorithm of Section 3.1 finds one. Moreover, the dimension of the generated ranking is minimal.*

There can be several reasons why a greedy algorithm could be incomplete. First, we could make a bad choice when selecting the transitions that are satisfied at a given dimension. However, there is no decision to make: if two transitions can be satisfied, one by a function σ_1, the other by a function σ_2, both can be simultaneously satisfied by the function $\sigma_1 + \sigma_2$. Second, enforcing that each transition is satisfied at the smallest possible dimension could also be a bad decision. Third, keeping all pairs $(\boldsymbol{x}, \boldsymbol{x}')$ in Inequality (4) until the transition is fully satisfied, even those for which the ranking is

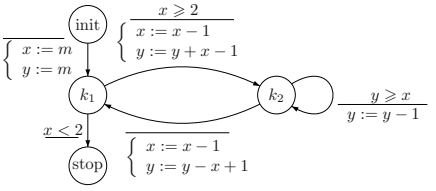
decreasing for a previous dimension, could overconstrain the problem too. In particular, asking that at least one transition is (fully) satisfied at each dimension (Line 5 of the algorithm) could be too demanding. One could imagine situations where all transitions are partially satisfied, but none is fully satisfied. Theorem 1 shows that this is not the case. Despite all these greedy choices, the completeness is not lost.

To summarize the proof, we start from an affine ranking of dimension d, if one exists. We show that there is an affine ranking of dimension d that fully satisfies at least one transition. This proves that our algorithm does not abort and generates a one-dimensional ranking σ. Then, we show that there is an affine ranking of dimension d whose first component is σ. Finally, we show that there is an affine ranking of dimension d, whose first component is σ, and such that the $d-1$ last components satisfy all transitions not fully satisfied by σ. Iterating the process, this shows our algorithm terminates and generates an affine ranking of dimension $\leq d$, for any possible dimension d.

The knowledgeable reader may have noticed a similarity with the algorithm of [13]. However, as pointed out earlier, the class of ranking functions we consider is larger. In our case, at each step of the construction, one component (i.e., dimension) σ of the global ranking function ρ is defined, and each control point k can have a different affine expression: $\sigma(k, \boldsymbol{x}) = \boldsymbol{\lambda}_k.\boldsymbol{x}+c_k$, where $\boldsymbol{\lambda}_k$ is a vector and c_k a scalar. The algorithm in [13] proceeds differently. At each step of the construction, instead of building a global ranking function, it checks, for each transition, if there exists an affine expression decreasing for this transition and non-increasing for all other transitions of the same strongly connected component (SCC). All transitions for which this is possible are removed as well as transitions that now do not belong to any SCC. One can prove that if this technique succeeds, there is also a component σ, which is decreasing for all removed transitions, non-increasing for other, of the form $\sigma(k, \boldsymbol{x}) = \boldsymbol{\lambda}.\boldsymbol{x} + c_k$, in other words a unique linear part for all control points, plus some shifts (the c_k), exactly as the loop scheduling technique of [18]. Such a restricted form is particularly useful when the automaton actions define simple translations, as for the example of Section 2.3, because Farkas lemma is then not needed. However, as the following examples show, this class of functions is less powerful than general affine rankings. In other words, the algorithm of [13] is not complete with respect to the class of all multi-dimensional affine rankings.

In the synthetic examples of Figure 2, to make the discussion simpler, we selected the lower bounds for x and y so that these two variables are always nonnegative. The two examples have then similar ranking functions: $2 + 3m$ and 0 for the start and stop program points, and $2x + y + 1$ for k_1, $x + y + 1$ for k_2 and k_3 (for the second example). They are thus proved to terminate with $O(m)$ transitions in any execution trace. If the same linear part is chosen in each SCC as previously explained (or equivalently if the technique of [13] is applied to prove termination), the result is not as accurate. The first component of the ranking cannot depend on y (due to the potentially-parametric increases and decreases of y on the two transitions between k_1 and k_2), it is thus a function of x only. For the first example, a two-dimensional ranking is generated (we get $x+1$ for k_1 and $(x+1, y)$ for k_2), thus the program is still proved to terminate but appears to have a quadratic complexity. For the second example however, as x decreases on the two transitions between k_1 and k_2, but increases on the self-loop on k_3, no transition can be satisfied at the first dimension, and the technique fails to prove termination.

```
x = m;
y = m;
while(x>=2) {
    x--; y = y+x;
    while(y>=x && indet()) y--;
    x--; y = y-x;
}
```




```
x = m;
y = m;
while(x>=2) {
    x--; y = y+x;
    while(y>=x+1 && indet()) {
      y--;
      while(y>=x+3 && indet()) {
        x++; y = y-2;
      }
      y--;
    }
    x--; y = y-x;
}
```

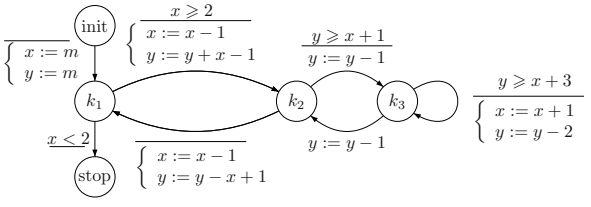


Fig. 2. Examples requiring general affine ranking functions

4 Worst-Case Computational Complexity (WCCC)

As shown in the survey by Wilhelm et al. [33], the computation of a worst-case execution time (WCET) is a highly complex affair, as it has to take into account the program, its data, and the processor on which it runs. Handling all these complexities is beyond the scope of this paper. Our aim is to evaluate an *abstract* WCET, as would be observed on a processor with a perfectly additive timing model, executing one automaton transition in unit time. We call this quantity the *worst-case computational complexity* of the program (WCCC). Such an estimate can be useful, for example as a template with unknown coefficients, to be fitted to actual measurements by a process of regression. It is also standard in high-level synthesis to need an upper-bound on the number of loop iterations (`do` loops as well as `while` loops), to enable scheduling optimizations at higher levels. We thus define the WCCC as an upper bound on the number of transitions executed, given an initial value of the counter variables. Note that the WCCC is significant only up to constant factors. For example, if we eliminate a state by edge coalescing, the semantics of the flowchart will not be materially changed, but the WCCC may decrease.

With this definition, one could over-approximate the WCCC of a terminating program by the total number of reachable states (because a finite trace cannot contain twice the same state), i.e., WCCC $\leq \sum_k \#\tilde{\mathcal{R}}_k$ or even more conservatively WCCC $\leq \sum_k \#\tilde{\mathcal{P}}_k$ as $\mathcal{R}_k$ is itself over-approximated by $\mathcal{P}_k$. [1] This is a very rough over-approximation but, even worse, this technique can lead to an infinite WCCC, even for a terminating automaton, if some invariant $\mathcal{P}_k$ is unbounded. Rather, we can use the ranking function itself to prune the invariant sets. Indeed, consider a trace $(k_0, \boldsymbol{x}_0)$, …, $(k_p, \boldsymbol{x}_p)$ in the execution of the automaton. By definition of a ranking function, $\rho(k_{i+1}, \boldsymbol{x}_{i+1}) \prec \rho(k_i, \boldsymbol{x}_i)$. Since $\prec$ is a strict order, it follows by transitivity that all $\rho(k_i, \boldsymbol{x}_i)$ are distinct in $\mathcal{W}$.

[1] Here, the notation $\tilde{\mathcal{S}}$ means the integral points in a set $\mathcal{S}$, and $\#\tilde{\mathcal{S}}$ denotes the cardinal of $\tilde{\mathcal{S}}$.

Hence, the length of the trace is bounded by the cardinal of the co-domain of ρ:

$$\text{WCCC} \leq \#\bigcup_k \rho(k, \tilde{\mathcal{P}}_k) \leq \sum_k \#\rho(k, \tilde{\mathcal{P}}_k) \tag{7}$$

The first inequality is more accurate but harder to compute as it involves a union of sets. So far, in our implementation, we use the second less accurate inequality.

Let us see how we can compute $\#\rho(k, \tilde{\mathcal{P}}_k)$ for a given control point k. To make notations simpler, we drop the index k: we let $\rho(k, \boldsymbol{x}) = \rho(\boldsymbol{x}) = R\boldsymbol{x} + \boldsymbol{r}$ and $\mathcal{P} = \mathcal{P}_k$. To compute $\#\rho(\tilde{\mathcal{P}})$, we can ignore the constant vector $\boldsymbol{r}$. The number of different values in $\rho(\tilde{\mathcal{P}})$ is then the number of points in the image of a $\mathbb{Z}$-polyhedron (intersection of an integral lattice, here $\mathbb{Z}^n$, and a polyhedron, here $\mathcal{P}$) by an affine function. If R is injective, it is of course equal to the number of integral points in the invariant itself. Otherwise, there are three issues: the fact that several points can have the same image (thus the kernel of the mapping must be identified), the fact that some regular holes can arise (sub-lattice of $\mathbb{Z}^n$) in the image of the polyhedron, and the fact that some irregular holes can appear on its boundaries. Such problems have been widely studied in the literature using various techniques related to Ehrhart polynomials [11,31]. So far, we implemented a simpler over-approximation method, which normalizes R in such a way that $\rho(\tilde{\mathcal{P}})$ no longer contains regular holes. This way, a standard computation of integral points can be applied. This normalization is done thanks to the Smith normal form. We compute $R = USV$, where U and V are unimodular, $S = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$, and D is a diagonal positive matrix of rank d (the rank of R). We compute $\mathcal{V}$ the polyhedron obtained by projecting the polyhedron $V\mathcal{P} = \{V\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{P}\}$ on its d first components. Actually, $\#\mathcal{V}$ is a slight over-approximation of $\#\rho(\tilde{\mathcal{P}})$. Indeed, two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\tilde{\mathcal{P}}$ have the same image by ρ if and only if $V\boldsymbol{x}$ and $V\boldsymbol{y}$ have the same d first components. The over-approximation comes from the fact that, in very specific cases, not all integral vectors in $\mathcal{V}$ are obtained by projection of an integral vector in $V\mathcal{P}$. The number of integral vectors in $\mathcal{V}$ is then computed using Ehrhart polynomials.

It is important to minimize the rank of D because the WCCC will tend to be smaller if the dimension of $\mathcal{V}$ is smaller. This is why it is important to generate rankings of minimal dimension as our algorithm does (Theorem 1). However, adding linearly-dependent components to the ranking will simply add null rows at the bottom of the matrix S. From this follows that the WCCC will be $O(M^n)$, if M is an upper bound for all variables, since it is impossible to build more than n linear forms on n variables. This bound cannot be improved, since with n variables, one can write a system of n perfectly nested loops, which achieves the required complexity.

The factors affecting the precision of the WCCC, beside the union computation, are the presence of non affine guards and of non affine domains. For example, the loop `for(j=1; j<m; j=2*j)` has invariant $2 \leq j < m$ (in the loop) and ranking j, which gives a WCCC of m instead of the correct value $\log_2 m$. Such a WCCC cannot be obtained by an affine technique, which grossly over-estimates the domain of j by a polyhedron. Another problem is that the invariant at a loop entry is often a coarse polyhedral approximation of a union of more accurate invariants on each path in the loop. Imposing the non-negative constraint (2) for such a control point is not necessary. It is enough to impose it for one control point per circuit of the automaton where invariants are more

accurate. Note also that, if one wants to count the number of loop traversals and not the number of transitions, it is not necessary to extend the sum in (7) to all nodes. For instance, if we include only one well-chosen state per loop, we will get a bound on the total number of loop traversals in an execution of the program.

Example of Section 2.3 (Cont'd). Inequality (7) gives the upper bound:

$$\text{WCCC} \leq \#\rho(\mathcal{P}_{start}) + \#\rho(\mathcal{P}_{lbl_4}) + \#\rho(\mathcal{P}_{lbl_5}) + \#\rho(\mathcal{P}_{lbl_6}) + \#\rho(\mathcal{P}_{lbl_{10}}) + \#\rho(\mathcal{P}_{stop})$$

Let us detail the computation of $\#\rho(\mathcal{P}_{lbl_4})$. $\rho(\mathcal{P}_{lbl_4}, x, y, m) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \end{bmatrix} (x\ y\ m)^T + \begin{bmatrix} 3 \\ 3 \end{bmatrix}$

Here, the mapping is bijective, so it would be sufficient to count the integral points in $\mathcal{P}_{lbl_4}$. But let us do the computation: $\begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \end{bmatrix} = U \times D \times V = \begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 6 & 0 \end{bmatrix} \times \begin{bmatrix} -2 & 3 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

Projecting $V\mathcal{P}_{lbl_4}$ along its first two dimensions amounts to consider the linear system: $\{p_1 = -2x+3y, p_2 = x-y, 0 < x \leq m, 0 < y \leq m\}$ and to eliminate every variable except p_1, p_2 and m (the parameter). This gives the polyhedron $\mathcal{V}$ defined by the constraints $\{1 \leq p_1 + 2p_2 \leq m, 1 \leq p_1 + 3p_2 \leq m\}$, whose number of points is an upper bound for $\#\rho(\tilde{\mathcal{P}}_{lbl_4})$. The cardinal of $\tilde{\mathcal{V}}$ is computed thanks to Ehrhart polynomials [10] (see Section 5). The result is, in general, a collection of polynomial formulas guarded by affine constraints on the parameters. Here, we get: $\#\rho(\mathcal{P}_{lbl_4}) \leq \#\mathcal{V} = m^2$ as expected. Applying the same process on the other control points, we finally obtain:

$$\text{WCCC} \leq 1 + (m^2) + (1 + 2m + m^2) + (2 + 3m + m^2) + (2m + m^2) + 1 = 5 + 7m + 4m^2.$$

5 Implementation and Experimental Results

We have built a tool suite that converts a C program into an integer interpreted automaton, constructs its invariants, tests its termination and, if successful, computes an upper bound for its worst-case computational complexity WCCC.

The first tool, c2fsm, turns a C program into an integer interpreted automaton, doing the relevant approximations when the program cannot be exactly translated. Our guidelines have been to consider only assignments to integer variables, and to give a variable an undefined value unless it is expressed as an affine form of integer variables. This tool also implements dead code elimination, useless variables elimination, and, as an option, the selection of cutpoints and the elimination of other control points. Note that it may be possible to extract flowcharts from binaries or assembly code, thus greatly extending the scope of the method.

The second tool, Aspic ([21], `http://laure.gonnord.org/aspic/`), a public-domain implementation of abstract acceleration [14], computes the invariants for every control point of the obtained integer interpreted automaton. Compared to the standard widening approach, this method computes a more precise reachability set for "acceleratable" loops, which locally avoids the use of widening and globally increases precision.

The third tool, Rank, implements the method described in this paper. Starting from the integer interpreted automaton and the invariants given by Aspic, Rank tries to prove the termination of the program by computing (multidimensional affine) ranking functions. In case of success, Rank computes the worst-case computational complexity of

the program. Also, in case of failure, Rank tries to exhibit a counterexample that causes non-termination. The linear programs involved in the termination part are solved thanks to the PIP tool (Parametric Integer Programming). The WCCC part requires counting the number of points into a $\mathbb{Z}$-polyhedron. This is done thanks to the Ehrhart polynomial part of the Polylib library (`http://icps.u-strasbg.fr/polylib`). The final result is a set of Ehrhart polynomials, guarded by affine predicates on program parameters.

On the web page `http://www.ens-lyon.fr/LIP/COMPSYS/Tools/Ranking/`, a table of experimental results can be found. Examples were collected from the extent literature, and notably from `http://www.eecs.qmul.ac.uk/~aziem/esop.html`. In all test cases we were able to prove termination, even for nondeterministic examples. Nested loops are correctly handled, and we find multi-dimensional rankings for them. WCCCs are returned by Rank as piecewise functions depending on the initial values of the variables: the table only provides the most general term of these expressions.

We were also able to prove the termination of some classical sorting algorithms. The rankings for these codes may seem of the wrong dimensions, but the additional dimensions have constant values and the orders of magnitude of the WCCC are still as expected, e.g., $O(N^2)$ for `bubblesort`. For `heapsort`, our algorithm finds an $O(N^2)$ WCCC instead of the correct $O(N \log_2 N)$, see Section 4 for an explanation.

Our tools are completely autonomous within the stated limitations on input programs. The precision of the results is strongly dependent on the quality of the invariants and of the affine approximation of some (non affine) affectations in the C programs. This is not a surprise as stated by Theorem 1: the quality of our technique is to be understood with respect to the quality of invariants that are provided.

6 Related Work

Our work establishes connections with at least three different techniques. First, it brings to the field of program termination, techniques primarily designed for scheduling and optimizing do loops, in the context of automatic parallelization [17]. The fundamental difference is that, for program termination, each problem dimension corresponds to an integer variable while, for automatic parallelization, it corresponds to a predefined loop counter. In this sense, it has also some similarities with the seminal work of Karp, Miller, and Winograd on systems of uniform recurrence equations [25]. Our algorithm to generate ranking functions is inspired by the algorithm of Feautrier [19] and its completeness [32] for scheduling affine loops. Counting techniques using Ehrhart polynomials are also standard for optimizing loops [11].

Second, it extends the ranking techniques previously proposed to prove the termination of programs. Using ranking functions to prove correctness was first proposed in [20]. Early approaches were semi-automatic: one had to guess ranking functions, and then prove their correctness using some form of Hoare logic. Attempts to automate this process started, first with one-dimensional linear rankings such as in [12,28], then with multi-dimensional rankings such as in [13,8], and propositions to build some forms of polynomial rankings followed [7,15]. Unlike ours, the techniques of Podelski and Rybalchenko [28] and of Bradley, Manna, and Sipma [8] are designed for "single-path linear loops", i.e., programs abstracted by an automaton with a single node. [28] formulates the constraints to get a one-dimensional ranking if it exists using Farkas lemma,

while [8] gives a complete method to derive, for a single node, a multi-dimensional ranking. It also tries to compute the invariants and the ranking functions simultaneously. Unlike these two methods, the technique of Colón and Sipma [13] handle flowchart programs of arbitrary structure. As explained in Section 3.2, the rankings it can generate correspond to a subclass of affine rankings where all control points within the strongly connected component being considered have the same linear part. It is not complete for the class of general multi-dimensional affine ranking functions, as the examples of Section 3.2 demonstrate. Finally, none of these techniques has been designed or extended to compute upper bounds on the WCCC, i.e., the maximal length of an execution trace.

To summarize, we extend previous work on affine ranking functions in several directions. First, unlike [28,8], we are not limited to one loop, i.e., our automaton can have an arbitrary number of vertices (as in reference [13]). As shown by the example of Section 2.3, this is mandatory to analyze complex loops, either nested loops, or multi-path simple loops that have been transformed into an automaton with several vertices by path-sensitive analysis. Second, to decide at which dimension of the ranking function a transition decreases (it must be non-increasing for the previous components of the ranking), the algorithm `has_llrf` in [8, Figure 2] is a potentially-exponential recursive exploration. Since the algorithm is also potentially exhaustive, there is no need to prove completeness. In contrast, as our algorithm is greedy, a completeness proof is needed, which is also an order of magnitude more general since we deal with the much larger space of all multi-dimensional affine ranking functions, not just one single lexicographic function. Third, unlike previous papers, we are able to prove that we get the smallest number of dimensions for each ranking function. In [7], the authors do notice that they may have as many dimensions as the number of transitions. As explained in Section 4, this dimension reduction is important for the computation of the WCCC.

In a different context, a large body of research followed the introduction of the size change termination (SCT) principle in [26]. The difference in the two approaches are first in semantics: the automaton represents a call graph instead of a control graph, and the variables may be summary information about data structures, like the length of a list or the size of a tree. More importantly, the relations between input and output variables of a transition are restricted to one of the two forms $x' < y$ and $x' \leq y$. Attempts to relax this restriction can be found in [3,4,5]. Once a set of size change relations has been found, termination follows if one can combine them in such a way that one variable at least is guaranteed to decrease. Such a combination can be interpreted as a ranking function, albeit of a shape fairly different from ours. Algorithms are provided to derive rankings, with a high complexity (at least in theory) due to their combinatorial nature.

Another trend of research has been started in [29] and pursued in [9]. Here, one uses several (local) ranking relations, all of them well founded, the intuition being that each relation proves termination of a part of the program. A consistency condition is necessary: the transitive closure of the transition relation of the program must be included in the union of all local ranking relations. The problem is how to find the local rankings, and how to prove the consistency condition. It may be that we can help at least for the first problem: apply our algorithm to cleverly chosen subsets of the automaton states, as for example strongly connected components or loops. However, as pointed out in [24], how to use local rankings (instead of global ones) for WCCC computations is not clear.

The third and last connection with previous work is related to the WCCC computation. The method of Gulwani et al. [24,23] for proving termination and bounding the complexity consists in creating counters – new variables which are incremented when traversing some transitions – and asking an invariant generator for bounds on the counter values. An elaborate system is proposed for selecting the transitions to be counted, which necessitates repeated calls to the invariant generator. Our method is related to this work in the following way. After a first round of computation of ranking functions, let us create a new counter which is reset to zero at the beginning of the program and incremented at each transition satisfied by the first component of our ranking (transitions t for which the variable ε_t of Section 3.1 is equal to 1). By construction, at each control point, the sum of the counter value and the affine expression given by the ranking is non-increasing, which provides an affine bound for this counter. We can continue in this way as the construction of ranking functions progresses and transitions are removed, making sure that new counters are reset to zero at the entry to each program fragment (i.e., on incoming transitions that were previously satisfied). If and when all edges are satisfied, we have found a system of counters which meets the constraints of Gulwani et al. Hence, our approach can be seen as a replacement for the counter placement algorithm of [24]. Both techniques rely on abstract interpretation to build initial invariants. Our technique is then guaranteed to find an adequate placement of counters if one exists, given these initial invariants, while the approach of Gulwani et al. is dependent on the unavoidable approximations made in abstract interpretation to build new invariants including the counters. Which method is best from the point of view of practical complexity is difficult to ascertain, since we avoid calling the invariant generator many times, but at the price of having to solve much larger linear programming problems. However, we point out that, in [24], counters are placed only on particular transitions selected *a priori*, typically the back edges of the control-flow graph. But, in the example of Section 2.3, both back edges (the self-loop on lbl_6 and the transition from lbl_{10} to lbl_4) are traversed a quadratic number of times, so there is no transition to place a linearly-bounded counter and the algorithm of [24] would fail. As our ranking function shows, the "outer" counter should be placed either on the transition from lbl_5 to lbl_6 or on the transition from lbl_6 to lbl_{10}. Or the graph must be transformed as proposed in [23], but with a risk of complexity increase. We believe that our work bridges the gap between techniques based on the placement of counters and the use of abstract interpretation to bound them, and techniques based on *global* ranking functions to derive complexity bounds.

7 Conclusion

7.1 Contributions

The first main contribution of this paper is the design of an algorithm for the construction of multi-dimensional affine ranking functions, which, in contrast to the combinatorial algorithm of [7], is greedy but nevertheless complete (with respect to the invariants found and the class of ranking functions considered) and optimal in the dimension of the ranking function. The algorithm makes no assumption on the shape of the source

program, and can handle, with proper preprocessing (i.e., after the program is approximated to fit into the affine interpreted automaton model), multiple loops of arbitrary nesting patterns, premature termination and `goto`s, nondeterministic choices and values, exceptions, and affine guards of arbitrary structure. We also point out that, in case of failure, our algorithm may exhibit a certificate of non termination in the form of an execution trace which may not terminate. The computation of the worst-case computational complexity (WCCC) is delegated to a very comprehensive stand-alone algorithm. This means that no arbitrary restrictions about the shape of loops and tests are necessary. We can directly rely on existing methods and tools for counting integer points within $\mathbb{Z}$-polyhedra and images of $\mathbb{Z}$-polyhedra by affine functions.

More generally, our work establishes a strong link with computation models, theoretical results, and tools developed by the community of automatic parallelization and high-performance computing, which seem to be not so used (or partly re-discovered) in the context of program termination. We believe that this connection can lead to further fruitful advances in the solution of problems faced by both communities.

7.2 Future Work

There is nevertheless room for many improvements. The preprocessor we use for converting a program into an interpreted automaton is somewhat brute force: any construct that is not affine in integer variables is replaced by the bottom value, which is absorbing ($\bot \oplus x = \bot$ for most operators), and which prints as `true` in a guard and as a question mark in an action. This can be improved by noticing that some operations, like modulo and integer division, can be linearized by the introduction of fresh variables, or that a bottom value may be constrained: for instance, a square is always non-negative. Also, variables with a finite domain, like Booleans and enums, can be used to refine the states. This may result in a large increase in the size of the automaton but has the direct benefit of extending the class of ranking functions considered, as these do not need to be affine anymore for such "unrolled" variables. Making sure that domains of integer variables are "fat" (to use the terminology of [16]) increases the chance that an affine ranking exists and improves the quality of the WCCC produced.

There is always room for improving an invariant constructor like Aspic. One may for instance improve the acceleration algorithms and loops treatment, or use additional abstract interpretation frameworks, like the congruences and lattices of [22]. It may also be interesting to construct the invariants *on demand*, both to improve the accuracy and to reduce the overhead of the method.

Last but not least, the power of the ranking algorithm can be increased in many ways. For instance, imposing that ranking functions are nonnegative everywhere (see Inequality (2)) is too strong a constraint. It is enough to impose it at a set of cut points. If the automaton graph becomes acyclic when these cut points are removed, then termination is still guaranteed, notwithstanding the relaxed nonnegativity constraint. In a way, eliminating all states but cutpoints before computing a ranking (path coalescing) is equivalent to relaxing the positivity constraint, but it is obtained at the cost of a potential increase in the number of transitions: if the eliminated state has n ingoing and m outgoing transitions, its elimination will generate up to $n \times m$ transitions. We still need to explore this trade-off and analyze its consequences on the WCCC computations.

Research on the SCT paradigm has shown that ranking functions of a more complex shape, like piecewise affine functions, are necessary in some cases. In our framework, this means splitting the invariant of some state(s) by an affine constraint. How to choose the states to split and the splitting predicate is left for future research.

A point we have not investigated is the termination of distributed programs. Our algorithm fails when termination depends on a fairness hypothesis.

References

1. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Bounding the computational complexity of flowchart programs with multi-dimensional rankings. Research Report 7235, INRIA (March 2010)
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L., Quinson, C.: Program termination and worst-time complexity with multi-dimensional affine ranking functions. Research Report 7037, INRIA (November 2009)
3. Anderson, H., Khoo, S.C.: Affine-based size-change termination. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 122–140. Springer, Heidelberg (2003)
4. Ben-Amram, A.M.: Size-change termination with difference constraints. ACM Transactions on Programming Languages and Systems (TOPLAS) 30(3), 1–31 (2008)
5. Ben-Amram, A.M.: Size change termination, monotonicity constraints, and ranking functions. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 109–123. Springer, Heidelberg (2009)
6. Blass, A., Gurevich, Y.: Inadequacy of computable loop invariants. ACM Transactions on Computational Logic (TOCL) 2(1), 1–11 (2001)
7. Bradley, A.A., Manna, Z., Sipma, H.B.: The polyranking principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005)
8. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
9. Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H.: Ranking abstractions. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 81–92. Springer, Heidelberg (2008)
10. Clauss, P.: Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In: International Conference on Supercomputing (ICS 1996), pp. 278–285. ACM, New York (1996)
11. Clauss, P.: Handling memory cache policy with integer points counting. In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 285–293. Springer, Heidelberg (1997)
12. Colón, M., Sipma, H.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001)
13. Colón, M.A., Sipma, H.B.: Practical methods for proving program termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
14. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5th ACM Symposium on Principles of Programming Languages (POPL 1978), pp. 84–96. Tucson (January 1978)
15. Cousot, P.: Proving program invariance and termination by parametric abstraction, Lagrangian relaxation, and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005)

16. Darte, A., Khachiyan, L., Robert, Y.: Linear scheduling is nearly optimal. Parallel Processing Letters 1(2), 73–81 (1991)
17. Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser, Basel (2000) ISBN 0-8176-4149-1
18. Darte, A., Vivien, F.: Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. International Journal of Parallel Programming 25(6), 447–496 (1997)
19. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. International Journal of Parallel Programming 21(6), 389–420 (1992)
20. Floyd, R.W.: Assigning meaning to programs. In: Schwartz, J.T. (ed.) Symposium on Applied Mathematics, vol. 19, pp. 19–32. A.M.S, Providence (1967)
21. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
22. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: Abramsky, S. (ed.) TAPSOFT 1991.LNCS, vol. 494, pp. 169–192. Springer, Heidelberg (1991)
23. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), pp. 375–385. ACM, Dublin (2009)
24. Gulwani, S., Mehra, K.K., Chilimbi, T.: SPEED: Precise and efficient static estimation of program computational complexity. In: 36th ACM Symposium on Principles of Programming Languages (POPL 2009), pp. 127–139. Savannah (January 2009)
25. Karp, R.M., Miller, R.E., Winograd, S.: The organization of computations for uniform recurrence equations. Journal of the ACM 14(3), 563–590 (1967)
26. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. ACM SIGPLAN Notices 36(3), 81–92 (2001)
27. Manna, Z.: Mathematical Theory of Computing. McGraw-Hill, New York (1974)
28. Podelski, A., Rybalchenko, A.: In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
29. Podelski, A., Rybalchenko, A.: Transition invariants. In: Ganzinger, H. (ed.) IEEE Symposium on Logic in Computer Science (LICS 2004), pp. 32–41. IEEE Computer Society, Los Alamitos (July 2004)
30. Schrijver, A.: Theory of linear and integer programming. Wiley, NewYork (1986)
31. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Counting integer points in parametric polytopes using Barvinok's rational functions. Algorithmica 48(1), 37–66 (2007)
32. Vivien, F.: On the optimality of Feautrier's scheduling algorithm. Concurrency and Computation: Practice and Experience 15(11-12), 1047–1068 (2003); Euro-Par'02 Special Issue
33. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The determination of worst-case execution times—overview of the methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) 7(3), 1–53 (2008)