Ranking Function Synthesis for Bit-Vector Relations[⋆]

Byron Cook[1], Daniel Kroening[2], Philipp Rümmer[2], and Christoph M. Wintersteiger[3]

[1] Microsoft Research, UK
[2] Oxford University, UK
[3] ETH Zurich, Switzerland

Abstract. Ranking function synthesis is a key aspect to the success of modern termination provers for imperative programs. While it is well-known how to generate linear ranking functions for relations over (mathematical) integers or rationals, efficient synthesis of ranking functions for machine-level integers (bit-vectors) is an open problem. This is particularly relevant for the verification of low-level code. We propose several novel algorithms to generate ranking functions for relations over machine integers: a complete method based on a reduction to Presburger arithmetic, and a template-matching approach for predefined classes of ranking functions based on reduction to SAT- and QBF-solving. The utility of our algorithms is demonstrated on examples drawn from Windows device drivers.

1 Introduction

Modern termination provers for imperative programs compose termination arguments by repeatedly invoking ranking function synthesis tools. Such synthesis tools are available for numerous domains, including linear and non-linear systems, and data structures. Thus, complex termination arguments can be constructed that reason simultaneously about the heap as well as linear and non-linear arithmetic.

Efficient synthesis of ranking functions for machine-level bit-vectors, however, has remained an open problem. Today, the most common approach to create ranking functions over machine integers is to use tools actually designed for rational arithmetic. Because such tools do not faithfully model all properties of machine integers, it can happen that invalid ranking functions are generated (both for terminating and for non-terminating programs), or that existing ranking functions are not found. Both phenomena can lead to incompleteness of termination provers: verification of actually terminating programs might fail.

⋆ Supported by the Swiss National Science Foundation grant no. 200021-111687, by the Engineering and Physical Sciences Research Council (EPSRC) under grant no. EP/G026254/1, by the EU FP7 STREP MOGENTES, and by the EU ARTEMIS CESAR project.

This paper considers the termination problem as well as the synthesis of ranking functions for programs written in languages like ANSI-C, C++, or Java. Such languages typically provide bit-vector arithmetic over 16, 32, or 64 bit words, and usually support both unsigned and signed datatypes (represented using the 2's complement). We present two new algorithms to generate ranking functions for bit-vectors: (i) a complete method based on a reduction to Presburger arithmetic, and (ii) a template-matching approach for predefined classes of ranking functions, including an extremely efficient SAT-based method. We quantify the performance of these new algorithms using examples drawn from Windows device drivers. Our algorithms are compared to the linear ranking function synthesis engine Rankfinder, which uses rational arithmetic.

Programs using *only* machine integers can also be proved terminating without ranking functions. Therefore, we also compare the performance of our methods with one approach not based on ranking functions, the rewriting of termination properties to safety properties according to Biere et al. [5].

Our results indicate that, on practical examples, the presented new methods clearly surpass the known methods in terms of precision and performance.

This paper is organised as follows: in Sect. 2, we provide motivating examples, briefly explain the architecture of termination provers and define the set of considered programs. In Sect. 3, a known, linear programming based approach for ranking function synthesis is analysed. Subsequently, a new extension to this method is presented that handles bit-vector programs soundly. Sect. 3.3 presents two approaches based on template-matching for predefined classes of ranking functions. In Sect. 4, the results of an experimental evaluation of all new methods are given and compared to results obtained through known approaches.

2 Termination of Bit-Vector Programs

We start by discussing two examples extracted from Windows device drivers that illustrate the difficulty of termination checking for low-level code. Both examples will be used in later sections to illustrate our methods.

The first example (Fig. 1) iterates for as many times as there are bits set in `i`. Termination of the loop can be proven by finding a *ranking function*, which is a function into a well-founded domain that monotonically decreases in each loop iteration. To find a ranking function for this example, it is necessary to take the semantics of the bit-wise AND operator `&` into account, which is not easily possible in arithmetic-based ranking function synthesis tools (see Sect. 3.1). A possible ranking function is the linear function $m(\mathtt{i}) = \mathtt{i}$, because the result of `i & (i-1)` is always in the range $[0, \mathtt{i}-1]$: the value of $m(\mathtt{i})$ decreases with every iteration, but it can not decrease indefinitely as it is bounded from below.

The second program (Fig. 2) is potentially non-terminating, because the variable `nLoop` might be initialised with a value that is not a multiple of 4, so that the loop condition is never falsified. For a correct analysis, it is necessary to know that integer underflows do not change the remainder modulo 4. Ignoring overflows, but given the information that the variable `nLoop` is in the

```
unsigned char i;
while (i!=0)
  i = i & (i-1);
```

Fig. 1. Code fragment of Windows driver kernel/agplib/init.c (#40 in our benchmarks).

```
unsigned long ulByteCount;
for (int nLoop = ulByteCount;
      nLoop; nLoop -= 4) { [...] }
```

Fig. 2. Code fragment of Windows device driver audio/gfxswap.xp/filter.cpp (#14 in our benchmarks).

range $[-2^{31}, 2^{31}-1]$ and is decremented in every iteration, a ranking function synthesis tool might incorrectly produce the ranking function `nLoop`.

2.1 Syntax and Semantics of Bit-Vector Programs

In order to simplify presentation, we abstract from the concrete language and datatypes and introduce a simpler category of bit-vector programs. Real-world programs can naturally be reduced to our language, which is in practice done by the Model Checker (possibly also taking care of data abstractions, etc).

We assume that bit-vector programs consist of only a single loop (endlessly repeating its body), possibly preceded by a sequence of statements (the *stem*).[4] Apart from this, our program syntax permits guards ($\mathsf{assume}\ (t)$), sequential composition ($\beta; \gamma$), choice ($\beta \,\square\, \gamma$), and assignments ($x := t$). Programs operate on global variables $x \in \mathcal{X}$, each of which ranges over a set $\mathbb{B}^{\alpha(x)}$ of bit-vectors of width $\alpha(x) > 0$. The syntactic categories of programs, statements, and expressions are defined by the following grammar:

$$
\begin{array}{lcl}
\langle \mathit{Prog} \rangle & ::= & \langle \mathit{Stmt} \rangle\ \mathsf{repeat}\ \{\ \langle \mathit{Stmt} \rangle\ \} \\
\langle \mathit{Stmt} \rangle & ::= & \mathsf{skip} \mid \mathsf{assume}\ (\langle \mathit{Expr} \rangle) \mid \langle \mathit{Stmt} \rangle ; \langle \mathit{Stmt} \rangle \mid \langle \mathit{Stmt} \rangle \,\square\, \langle \mathit{Stmt} \rangle \mid x := \langle \mathit{Expr} \rangle \\
\langle \mathit{Expr} \rangle & ::= & 0_n \mid 1_n \mid \cdots \mid *_n \mid x \mid \mathsf{cast}_n(\langle \mathit{Expr} \rangle) \mid \neg \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle \circ \langle \mathit{Expr} \rangle
\end{array}
$$

Because the width of variables is fixed and does not change during program execution, it is not necessary to introduce syntax for variable declarations. Expressions $0_n, 1_n, \ldots$ are bit-vector literals of width n, the expression $*_n$ nondeterministically returns an arbitrary bit-vector of width n, and the operator cast_n changes the width of a bit-vector (cutting off the highest-valued bits, or filling up with zeros as highest-valued bits). The semantics of bitwise negation $\neg$, and of the binary operators $\circ \in \{+, \times, \div, =, <_s, <_u, \ \&\ , \ |\ , \ll, \gg\}$ is as

[4] This is not a restriction, as will become clear in the next section.

usual.[5] When evaluating the arithmetic operators $+, \times, \div, \ll, \gg$, both operands are interpreted as unsigned integers. In the case of the strict ordering relation $<_s$ (resp., $<_u$) the operands are interpreted as signed integers in 2's complement format (resp., as unsigned integers).

We write $t : n$ to denote that the expression t is correctly typed and denotes a bit-vector of width n. In the rest of the paper, we always assume that programs are type-correct.

The state space of programs defined over a (finite) set $\mathcal{X}$ of bit-vector variables with widths α is denoted by $\mathcal{S}$, and consists of all mappings from $\mathcal{X}$ to bit-vectors of the correct width: $\mathcal{S} = \{f \in \mathcal{X} \to \mathbb{B}^+ \mid f(x) \in \mathbb{B}^{\alpha(x)} \text{ for all } x \in \mathcal{X}\}$. The transition relation defined by a statement β is denoted by $R_\beta \subseteq \mathcal{S} \times \mathcal{S}$. In particular, we define the transition relation for sequences as $R_{\beta_1;\beta_2}(s, s') \equiv \exists s'' \, . \, R_{\beta_1}(s, s'') \wedge R_{\beta_2}(s'', s')$.

Example. We consider the program given in Fig. 2. Using unsigned arithmetic (and $-4 \equiv 2^{32} - 4 \mod 2^{32}$), the bit-vector program for a single loop iteration is

$$\mathsf{assume}\ (\mathit{nLoop} \neq 0);\ \mathit{nLoop} := \mathit{nLoop} + (2^{32} - 4) \tag{1}$$

Complexity. We say that a bit-vector program β repeat $\{\ \gamma\ \}$ *terminates* if there is no infinite sequence of states $a_0, a_1, a_2, \ldots \in \mathcal{S}$ with $R_\beta(a_0, a_1)$ and $R_\gamma(a_i, a_{i+1})$ for all $i > 0$. The termination problem for bit-vector programs is decidable:

Lemma 1. *Deciding termination of bit-vector programs is PSPACE-complete in the program length*[6] *plus* $\sum_{x \in \mathcal{X}} \alpha(x)$*, i.e., the size of the program's available memory.*

Practically, the most successful termination provers are based on incomplete methods that try to avoid this high complexity, by such means as the generation of specific kinds of ranking functions (like functions that are linear in program variables). The general strategy of such provers is described in the next section.

2.2 Binary Reachability Analysis and Ranking Functions

Definition 1 (Ranking function). *Suppose* $(D, \prec)$ *is a well-founded, strictly partially ordered set, and* $R \subseteq U \times U$ *is a relation over a non-empty set* U*. A ranking function for* R *is a function* $m : U \to D$ *such that:*

$$\text{for all } a, b \in U : R(a, b) \text{ implies } m(b) \prec m(a).$$

Of particular interest in the context of this paper is the well-founded domain of natural numbers $(\mathbb{N}, <)$. In general, we can directly conclude:

Lemma 2. *If a (global) ranking function exists for the transition relation* R *of a program* β*, then* β *terminates.*

[5] Adding further operations, e.g., bit-vector concatenation, is straightforward.

[6] The number of characters in the program text. We assume that a unary representation is used for the index n of the operators $0_n, 1_n, \ldots, *_n$, and cast_n.

The problem of deciding termination of a program may thus be stated as a problem of ranking function synthesis. By the disjunctive well-foundedness theorem [15], this is simplified to the problem of finding a ranking function for every *path* through the program. The ranking functions found for all n paths are used to construct a global, disjunctive ranking relation $M(a,b) = \bigvee_{i=1}^{n} m_i(b) \prec m_i(a)$.

A technique that puts this theorem to use is *Binary Reachability Analysis* [8,9]. In this approach, termination of a program is first expressed as a *safety* property [5], initially assuming that the program does *not* terminate. Consequently, a (software) Model Checker is applied to obtain a counterexample to termination, i.e., an example of non-termination. This counterexample contains a *stem* that describes how to reach a loop in the program, and a *cycle* that follows a path π through the loop, finally returning to the entry location of the loop. What follows is an analysis solely concerned with the stem and π, which is why we may safely restrict ourselves to single-loop programs here.

The next step in the procedure is to synthesise a ranking function for π, which can be seen as a new, smaller, and loop-free program that does not contain choice operators. Semantically, π is interpreted as a relation $R_\pi(x,x')$ between program states x, x'. If a ranking function m_π is found for this relation, the original safety property is weakened to exclude all paths of the program that satisfy the ranking relation $m_\pi(x') \prec m_\pi(x)$, and the process starts over. If no further non-terminating paths are found, termination of the program is proven.

3 Ranking Functions for Bit-Vector Programs

We introduce new methods based on integer linear programming, SAT-solving, and QBF-solving to synthesise ranking functions for paths in a bit-vector program. Before that, we give a short overview of the derivation of ranking functions using linear programming, which is the starting point for our methods.

3.1 Synthesis of Ranking Functions by Linear Programming

The approach to generate ranking functions that is used in binary reachability engines like Terminator [9] and ARMC [16] was developed by Podelski et al. [14]. In this setting, ranking functions are generated for transition relations $R \subseteq \mathbb{Q}^n \times \mathbb{Q}^n$ that are described by systems of linear inequalities:

$$R(x,x') \equiv Ax + A'x' \leq b \qquad (A, A' \in \mathbb{Q}^{k \times n}, b \in \mathbb{Q}^k)$$

where $x, x' \in \mathbb{Q}^n$ range over vectors of rationals. Bit-vector relations have to be encoded into such systems, which usually involves an over-approximation of program behaviour. The derived ranking functions are linear and have the codomain $D = \{z \in \mathbb{Q} \mid z \geq 0\}$, which is ordered by $y \prec z \equiv y + \delta \leq z$ for some rational $\delta > 0$. Ranking functions $m : \mathbb{Q}^n \to D$ are represented as $m(x) = rx + c$, with $r \in \mathbb{Q}^n$ a row vector and $c \in \mathbb{Q}$. Such a function m is a ranking function with the domain $(D, \prec)$ if and only if the following condition holds:

$$\text{for all } x, x' \in \mathbb{Q}^n : R(x,x') \text{ implies } rx + c \geq 0 \wedge rx' + c \geq 0 \wedge rx' + \delta \leq rx \quad (2)$$

Coefficients r for which this implication is satisfied can be constructed using Farkas' lemma, of which the 'affine' form given in [19] is appropriate. Using this lemma, a necessary and sufficient criterion for the existence of linear ranking functions can be formulated:

Theorem 1 (Existence of linear ranking functions [14]). *Suppose that $R(x, x') \equiv Ax + A'x' \leq b$ is a satisfiable transition relation. R has a linear ranking function $m(x) = rx + c$ iff there are non-negative vectors $\lambda_1, \lambda_2 \in \mathbb{Q}^k$ s.t.:*

$$\lambda_1 A' = 0, \quad (\lambda_1 - \lambda_2)A = 0, \quad \lambda_2(A + A') = 0, \quad \lambda_2 b < 0.$$

In this case, m can be chosen as $\lambda_2 A'x + (\lambda_1 - \lambda_2)b$.

This criterion for the existence of linear ranking functions is necessary and sufficient for linear inequalities on the rationals, but only sufficient over the integers or bit-vectors: there are relations $R(x, x') \equiv Ax + A'x' \leq b$ for which linear ranking functions exist, but the criterion fails, e.g.:

$$R(x, x') \;\equiv\; x \in [0, 4] \wedge x' \geq 0.2x + 0.9 \wedge x' \leq 0.2x + 1.1 \;.$$

Restricting x and x' to the integers, this is equivalent to $x = 0 \wedge x' = 1$ and can be ranked by $m(x) = -x + 1$. Over the rationals, the program defined by the inequalities does not terminate, which implies that no ranking function exists and the criterion of Theorem 1 fails.

3.2 Synthesis of Ranking Functions by Integer Linear Programming

To extend the approach from Sect. 3.1 and fully support bit-vector programs, we first generalise Theorem 1 to disjunctions of systems of inequalities over the integers. We then define an algorithm to synthesise linear ranking functions for programs defined in Presburger arithmetic, which subsumes bit-vector programs.

Linear ranking functions over the integers. In order to faithfully encode bit-vector operations like addition with overflow (describing non-convex transition relations), it is necessary to consider also disjunctive transition relations R:

$$R(x, x') \;\equiv\; \bigvee_{i=1}^{l} A_i x + A'_i x' \leq b_i \tag{3}$$

where $l \in \mathbb{N}$, $A_i, A'_i \in \mathbb{Z}^{k \times n}, b_i \in \mathbb{Z}^k$, and $x, x' \in \mathbb{Z}^n$ range over integer vectors. Linear ranking functions for such relations can be constructed by solving an implication like (2) for each disjunct of the relation, as shown below. There is one further complication, however: Farkas' lemma, which is the main ingredient for Theorem 1, is in general not complete for inequalities over the integers.

Farkas' lemma is complete for *integral* systems, however: $Ax + A'x' \leq b$ is called integral if the polyhedron $\{\binom{x}{x'} \in \mathbb{Q}^{2n} \mid Ax + A'x' \leq b\}$ coincides with its integral hull (the convex hull of the integer points contained in it). Every system of inequalities can be transformed into an integral system with the same integer solutions, although this might increase the size of the system exponentially [19].

Lemma 3. *Suppose* $R(x,x') \equiv \bigvee_{i=1}^{l} A_i x + A'_i x' \leq b_i$ *is a transition relation in which each disjunct is satisfiable and integral. R has a linear ranking function* $m(x) = rx + c$ *if and only if there are non-negative vectors* $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^k$ *for* $i \in \{1, \ldots, l\}$ *such that:*

$$\lambda_1^i A'_i = 0, \quad \lambda_2^i (A_i + A'_i) = 0, \quad \lambda_2^i b_i < 0, \quad (\lambda_1^i - \lambda_2^i) A_i = 0, \quad \lambda_2^i A'_i = r. \tag{4}$$

Ranking functions for Presburger arithmetic. Presburger arithmetic (PA) is the first-order theory of integer arithmetic without multiplication [17]. We describe a complete procedure to generate linear ranking functions for PA-defined transition relations by reduction to Lem. 3.[7]

Suppose a transition relation $R(x,x')$ is defined by a Presburger formula. Because PA allows quantifier elimination [17], it can be assumed that $R(x,x')$ is a quantifier-free Boolean combination of equations, inequalities, and divisibility constraints $\epsilon \mid (cx + dx' + e)$. Divisibility constraints are introduced during quantifier elimination and state that the value of the term $cx + dx' + e$ (with $c, d \in \mathbb{Z}^n, e \in \mathbb{Z}$) is a multiple of the positive natural number $\epsilon \in \mathbb{N}^+$.

In order to apply Lem. 3, we eliminate divisibility constraints from $R(x,x')$ as explained in detail below. This is possible by introducing auxiliary program variables y, y': we will transform $R(x,x')$ to a formula $R'(x,y,x',y')$ without divisibility constraints, such that $\exists y, y'.R'(x,y,x',y') \equiv R(x,x')$. The transformation increases the size of the PA formula only polynomially.

By rewriting to disjunctive normalform, replacing equations $s = t$ with inequalities $s \leq t \wedge t \leq s$, the relation $R'(x,y,x',y')$ can be stated as in (3):

$$R'(x,y,x',y') \;\equiv\; \bigvee_{i=1}^{l} A_i \begin{pmatrix} x \\ y \end{pmatrix} + A'_i \begin{pmatrix} x' \\ y' \end{pmatrix} \leq b_i$$

We can then apply Lem. 3 to R' to derive a linear ranking function $m'(x,y)$. To ensure that no auxiliary variables y occur in $m'(x,y)$ (i.e., $m'(x,y) = m(x)$), equations are added to (4) that constrain the corresponding entries of r to zero.

Replacing divisibility constraints by disjunctions of equations. The following equivalences are used in the transformation from $R(x,x')$ to $R'(x,y,x',y')$:

$$\epsilon \mid (cx + dx' + e) \;\equiv\; \epsilon \;\Big|\; \Big(cx - \epsilon\Big\lfloor \frac{cx}{\epsilon} \Big\rfloor + dx' - \epsilon\Big\lfloor \frac{dx'}{\epsilon} \Big\rfloor + e\Big) \tag{5}$$

$$\equiv \bigvee_{\substack{i \in \mathbb{Z} \\ 0 \leq i\cdot\epsilon - e < 2\epsilon}} i \cdot \epsilon - e \;=\; cx - \epsilon\Big\lfloor \frac{cx}{\epsilon} \Big\rfloor + dx' - \epsilon\Big\lfloor \frac{dx'}{\epsilon} \Big\rfloor \tag{6}$$

$$\equiv \exists y_c, y'_d. \begin{pmatrix} 0 \leq cx - \epsilon y_c < \epsilon \wedge 0 \leq dx' - \epsilon y'_d < \epsilon \\ \wedge \left(\bigvee_{0 \leq i\cdot\epsilon - e < 2\epsilon} i \cdot \epsilon - e = cx - \epsilon y_c + dx' - \epsilon y'_d\right) \end{pmatrix} \tag{7}$$

[7] The procedure can also derive ranking functions that contain integer division expressions $\lfloor \frac{t}{\epsilon} \rfloor$ for some $\epsilon \in \mathbb{Z}$, but it is not complete for such functions. Assuming that a polynomial method is used to solve (4), the complexity of our procedure is singly exponential.

Equivalence (5) holds because divisibility is not affected by subtracting multiples of ϵ on the right-hand side, while (6) expresses that the value of the term $cx - \epsilon\lfloor \frac{cx}{\epsilon} \rfloor + dx' - \epsilon\lfloor \frac{dx'}{\epsilon} \rfloor$ lies in the right-open interval $[0, 2\epsilon)$. Therefore, the divisibility constraints of (5) are equivalent to a disjunction of exactly two equations. Finally, the integer division expressions $\lfloor \frac{cx}{\epsilon} \rfloor$ can equivalently be expressed using existential quantifiers in (7).

To avoid the introduction of new quantifiers, the quantified variables y_c, y'_d are treated as program variables. Whenever a constraint $\epsilon \mid (cx + dx' + e)$ occurs in $R(x, x')$, we introduce new pre-state variables y_c, y_d and post-state variables y'_c, y'_d that are defined by adding conjuncts to $R(x, x')$:

$$\begin{aligned} R'(x, y_c, y_d, x', y'_c, y'_d) \equiv \;\; & R(x, x') \wedge 0 \leq cx - \epsilon y_c < \epsilon \wedge 0 \leq dx - \epsilon y_d < \epsilon \\ & \wedge 0 \leq cx' - \epsilon y'_c < \epsilon \wedge 0 \leq dx' - \epsilon y'_d < \epsilon \end{aligned}$$

In $R'(x, y_c, y_d, x', y'_c, y'_d)$, the constraint $\epsilon \mid (cx + dx' + e)$ can then be replaced with a disjunction $\bigvee_{0 \leq i \cdot \epsilon - e < 2\epsilon} i \cdot \epsilon - e = cx - \epsilon y_c + dx' - \epsilon y'_d$ as in (7). Iterating this procedure eventually leads to a transition relation $R'(x, y, x', y')$ without divisibility judgements, such that $\exists y, y'. R'(x, y, x', y') \equiv R(x, x')$.

Representation of bit-vector operations in PA. Presburger arithmetic is expressive enough to capture the semantics of all bit-vector operations defined in Sect. 2, so that ranking functions for bit-vector programs can be generated using the method from the previous section. For instance, the semantics of a bit-vector addition $s + t$ can be defined in weakest-precondition style as:

$$wp(x := s + t, \phi) \;=\; wp\begin{pmatrix} y_1 := s;\; y_2 := t, \\ \exists x.(0 \leq x < 2^n \wedge 2^n \mid (x - y_1 - y_2) \wedge \phi) \end{pmatrix}$$

where $s : n, t : n$ denote bit-vectors of length n, and y_1, y_2 are fresh variables. The existentially quantified formula assigns to x the remainder of $y_1 + y_2$ modulo 2^n.

A precise translation of non-linear operations like $\times$ and $\&$ can be done by case analysis over the values of their operands, which in general leads to formulae of exponential size, but is well-behaved in many cases that are practically relevant (e.g., if one of the operands is a literal). Such an encoding is only possible because the variables of bit-vector programs range over finite domains of fixed size.

Example. We encode the bit-vector program (1) corresponding to Fig. 2 in PA:

$$\begin{aligned} & \mathit{nLoop} \neq 0 \;\wedge\; 2^{32} \mid (\mathit{nLoop}' - \mathit{nLoop} - 2^{32} + 4) \\ \wedge\; & 0 \leq \mathit{nloop} < 2^{32} \;\wedge\; 0 \leq \mathit{nloop}' < 2^{32} \end{aligned}$$

From the side conditions, we can read off that the term $\mathit{nLoop}' - \mathit{nLoop} - 2^{32} + 4$ has the range $[5 - 2^{33}, 3]$, so that the divisibility constraint can directly be split into two equations (auxiliary variables as in (7) are unnecessary in this particular example). With further simplifications, we can express the transition relation as:

$$\begin{aligned} & \big(\mathit{nLoop}' = \mathit{nLoop} - 4 \;\wedge\; 0 \leq \mathit{nloop}' \;\wedge\; \mathit{nloop} < 2^{32}\big) \\ \vee\; & \big(\mathit{nLoop}' = \mathit{nLoop} + 2^{32} - 4 \wedge 0 < \mathit{nloop} \wedge \mathit{nloop}' < 2^{32}\big) \end{aligned}$$

It is now easy to see that each disjunct is satisfiable and integral, which means that Lem. 3 is applicable. Because the conditions (4) are not simultaneously satisfiable for all disjuncts, no linear ranking function exists for the program.

3.3 Synthesis of Ranking Functions from Templates

A subset of the ranking functions for bit-vector programs can be identified by templates of a desired class of functions with undetermined coefficients. In order to find the coefficients, we consider two methods: (i) an encoding into quantified Boolean formulas (QBF) to check all suitable values, and (ii) a propositional SAT-solver to check likely values.

We primarily consider linear functions of the program variables. Let $x = (x_1, \ldots, x_{|\mathcal{X}|})$ be a vector of program variables and associate a coefficient c_i with each $x_i \in \mathcal{X}$. The coefficients constitute the vector $c = (c_1, \ldots, c_{|\mathcal{X}|})$. We can then construct the template polynomial

$$p(c, x) := \sum_{i=1}^{|\mathcal{X}|} (c_i \times \mathsf{cast}_w(x_i))$$

with the bit-width $w \geq \max_i(\alpha(x_i)) + \lceil \log_2(|\mathcal{X}| + 1) \rceil$ and $\alpha(c_i) = w$, chosen such that no overflows occur during summation. The following theorem provides a bound on w that guarantees that ranking functions can be represented for all programs that have linear ranking functions.

Theorem 2. *There exists a linear ranking function on path π with transition relation $R_\pi(x, x')$, if*

$$\exists c \; \forall x, x' \; . \; R_\pi(x, x') \Rightarrow p(c, x') <_s p(c, x) \; . \tag{8}$$

Vice versa, if there exists a linear ranking function for π, then Eq. (8) must be valid whenever

$$w \geq \max_i(\alpha(x_i)) \cdot (|\mathcal{X}| - 1) + |\mathcal{X}| \cdot \log_2 |\mathcal{X}| + 1 \; .$$

It is straightforward to flatten Eq. (8) into QBF. Thus, a QBF solver that returns an assignment for the top-level existential variables is able to compute suitable coefficients. Examples of such solvers are Quantor [4], sKizzo [3], and Squolem [13]. In our experiments, we use an experimental version of QuBE [11].

Despite much progress, the capacity of QBF solvers has not yet reached the level of propositional SAT solvers. We therefore consider the following simplistic way to enumerate coefficients: we restrict all coefficients to $\alpha(c_i) = 2$ and we fix a concrete assignment $\gamma(c) \in \{0, 1, 3\}$ to the coefficients (corresponding to $\{-1, 0, 1\}$ in 2's complement). Negating and applying γ transforms Equation 8 into

$$\neg \exists x, x' \; . \; R_\pi(x, x') \wedge \neg(p(\gamma(c), x') <_s p(\gamma(c), x)) \; , \tag{9}$$

which is a bit-vector (or SMT-$\mathcal{B}V$) formula that may be flattened to a purely propositional formula in the straightforward way. The formula is satisfiable iff

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**Loop**
list	list	unr.	i++	unr.	unr.	unr.	unr.	wait	unr.	unr.	i++	list	**Type**
126	85	687	248	340	298	253	844	109	375	333	3331	146	**CE Time [sec]**
0.5	0.1	–	0.7	–	–	–	–	0.4	–	–	2.2	0.4	**Synth. Time [sec]**
×	×	✓	MO	✓	✓	✓	✓	×	✓	✓	MO	×	**Terminates?**

Table 1. The behaviour on the loops of a keyboard driver.

p is *not* a genuine ranking function. Thus, we enumerate all possible γ until we find one for which Equation 9 is unsatisfiable, which means that $p(\gamma(c), x)$ must be a genuine ranking function on π. Even though there are $3^{|\mathcal{X}|}$ possible combinations of coefficient values to test, this method performs surprisingly well in practice, as demonstrated by our experimental evaluation in Sect. 4.

Example. We consider the program given in Fig. 1. The only variable in the program is i, and it is 8 bits wide. We construct the polynomial $p(c, i) = c \times \mathsf{cast}_9(i)$ with $\alpha(c) = 9$. For the only path through the loop in this example, the transition relation $R_\pi(i, i')$ is $i \neq 0 \wedge i' = i \ \& \ (i - 1)$. Solving the resulting formula

$$\exists c \forall i, i' \ . \ R_\pi(i, i') \Rightarrow p(c, i') <_s p(c, i)$$

with a QBF-Solver does not return a result within an hour. We thus rewrite the formula according to Equation 9 and obtain

$$\neg \exists i, i' \ . \ R_\pi(i, i') \wedge \neg(p(c, i') <_s p(c, i))$$

which we solve (in a negligible amount of runtime) for all choices of $c \in \{0, 1, 3\}$. The formula is unsatisfiable for $c = 1$, and we conclude that $\mathsf{cast}_9(i)$ is a suitable ranking function. In this particular example, it is possible to omit the cast.

4 Experiments

4.1 Large-scale benchmarks

Following Cook et al. [9], we implemented a binary reachability analysis engine to evaluate our ranking synthesis methods. Our implementation uses SATABS as the reachability checker [7], which implements SAT-based predicate abstraction. Our benchmarks are device drivers from the Windows Driver Development Kit (WDK).[8] The WDK already includes verification harnesses for the drivers. We use GOTO-CC[9] to extract model files from a total of 87 drivers in the WDK.

Most of the drivers contain loops over singly and doubly-linked lists, which require an arithmetic abstraction. This abstraction can be automated by existing shape analysis methods (e.g., the one recently presented by Yang et al. [20]).

[8] Version 6, available at `http://www.microsoft.com/whdc/devtools/wdk/`

[9] `http://www.cprover.org/goto-cc/`

Slicing the input. Just like Cook et al. [9], we find that most of the runtime is spent in the reachability checker (more than 99%), especially after all required ranking functions have been synthesised and no more counterexamples exist. To reduce the resource requirements of the Model Checker, our binary reachability engine analyses each loop separately and generates an inter-procedural slice [12] of the program, slicing backwards from the termination assertion. In addition, we rewrite the program into a single-loop program, abstracting from the behaviour of all other loops.[10] With this (abstracting) slicer in place, we find that absolute runtime and memory requirements are reduced dramatically.

As our complete data on Windows drivers is voluminous, we present a typical example in detail. The full dataset is available online.[11] The keyboard class driver in the WDK (KBDCLASS) contains a total of 13 loops in a harness (SDV_FLAT_HARNESS) that calls all dispatch functions nondeterministically.

Table 1 describes the behaviour of our engine on this driver. For every loop we list the type (list iteration, i++, unreachable, or 'wait for device'), the time it takes to find a potentially non-terminating path ('CE Time'), the time required to find a ranking function using our SAT template from Sect. 3.3 ('Synth. Time', where applicable), and the final result. In the last row, 'MO' indicates a memory-out after consuming 2 GB of RAM while proving that no further counterexamples to termination exist. The entire analysis of this driver requires 2 hours.[12]

We were able to isolate a possible termination problem in the USB driver bulkusb that may result in the system being blocked. The driver requests an interface description structure for every device available by calling an API function. It increments the loop counter if this did not return an error. The API function, however, may return NULL if no interface matches the search criteria, resulting in the loop counter not being incremented. Since numberOfInterfaces is a local (non-shared) variable of the loop, the problem would persist in a concurrent setting, where a device may be disconnected while the loop is executed.

4.2 Experiments on smaller examples

The predominant role of the reachability engine on our large-scale experiments prevents a meaningful comparison of the utility of the various techniques for ranking function synthesis. For this reason, we conducted further experiments on smaller programs, where the behaviour of the reachability engine has less impact. We manually extracted 61 small benchmark programs from the WDK drivers. Most of them contain bit-vector operations, including multiplication, and some of them contain nested loops. All benchmarks were manually sliced by

[10] Following the hypothesis that loop termination seldom depends on complex variables that are possibly calculated by other loops, our slicing algorithm replaces all assignments that depend on five or more variables with non-deterministic values, and all loops other than the analysed one with program fragments that havoc the program state (non-deterministic assignments to all variables that might change during the execution of the loop).

[11] `http://www.cprover.org/termination/`

[12] All experiments were run on 8-core Intel Xeon 3 GHz machines with 16 GB of RAM.

#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Manual Insp.	L	L	L	L	N	N	N	L	T	N	T	L	L	N	T	L	L	L	L	L	T	L	L	L	L	L	L	N	T	L	T
SAT	●	●	●	●	○	○	○	●	○	○	○	●	●	○	○	●	●	●	●	●	○	●	●	●	●	●	○	○	○	●	○
Seneschal	●	●	●	●	○	○	–	●	○	○	○	●	●	○	○	●	●	●	●	●	○	–	●	●	●	●	○	○	○	–	○
Rankfinder	○	●	○	●	○	○	○	●	◐	○	○	●	○	◐	◐	●	○	●	●	○	○	○	●	●	●	●	◐	○	–	○	○
QBF [-1,+1]	–	–	●	●	○	○	–	–	–	○	–	●	–	–	–	–	–	–	–	–	–	–	●	–	–	–	–	○	–	●	○
QBF $P(c,x)$	–	–	●	●	–	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–
Biere et al. [5]	–	–	–	●	–	–	–	–	–	○	–	●	●	–	–	–	–	–	–	●	–	–	●	–	–	–	–	○	–	–	●

#	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	
Manual Insp.	T	L	L	N	L	T	L	L	L	L	L	L	N	T	L	L	T	T	T	L	T	T	N	L	L	L	L	L	N	T	
SAT	○	●	○	○	●	○	●	●	●	●	●	●	○	○	●	●	○	○	○	●	○	○	○	●	●	●	●	●	○	○	
Seneschal	○	●	○	○	●	○	●	●	●	●	●	●	○	○	●	●	○	○	○	●	○	○	○	●	●	●	●	–	○	–	
Rankfinder	○	●	○	○	●	◐	●	●	○	○	●	●	○	○	●	●	○	○	○	●	–	○	–	●	○	●	●	○	○	○	
QBF [-1,+1]	○	–	–	–	–	–	–	–	●	–	–	●	○	–	–	●	–	○	○	●	○	–	–	–	–	–	–	–	○	–	
QBF $P(c,x)$	–	–	–	–	–	–	–	–	–	–	–	–	○	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–	–	–	
Biere et al. [5]	–	–	–	–	●	–	–	–	●	–	–	–	–	●	●	●	–	–	●	●	–	–	–	–	–	–	–	–	○	●	

● –	Termination was proven	T –	Terminating (non-linear)
○ –	(Possibly) Non-terminating	L –	Terminating, and linear ranking functions exist.
◐ –	Incorrect under bit-vector semantics		
– –	Memory or time limits exhausted	N –	Non-terminating

Table 2. Experimental results on 61 benchmarks drawn from Windows device drivers.

removing all source code that does not affect program termination (much like an automated slicer, but more thoroughly). We also employ the same abstraction technique as described in the previous section. All but ten of the benchmark programs terminate. The time limit in these benchmarks was $3600\,s$, and the memory consumption was limited to 2 GB.

To evaluate the integer linear programming method described in Sect. 3.2, we developed the prototype Seneschal.[13] It is based on the prover Princess [18] for Presburger arithmetic with uninterpreted predicates and works by (i) translating a given bit-vector program into a PA formula, (ii) eliminating the quantifiers in the formula, (iii) flattening the formula to a disjunction of systems of inequalities, and (iv) applying Lem. 3 to compute ranking functions. Seneschal does currently not, however, transform systems of inequalities to integral systems, which means that it is a sound but incomplete tool; the experiments show that transformation to integral systems is unnecessary for the majority of the considered programs.

Table 2 summarizes the results. The first column indicates the result obtained by manual inspection, i.e., if a specific benchmark is terminating, and if so whether there is a linear ranking function to prove this. The other columns represent the following ranking synthesis approaches: SAT is the coefficient enumeration approach from Sect. 3.3; Seneschal is the integer linear programming approach from Sect. 3.2; Rankfinder is the linear programming approach over rationals from Sect. 3.1; QBF [-1,+1] is a QBF template approach from Sect. 3.3 with coefficients restricted to $[-1,+1]$, such that the template represents the same ranking functions as the one used for the SAT enumeration approach. QBF $P(c,x)$ is the unrestricted version of this template. Note that two benchmarks (#27 and #34) are negatively affected by our slicer: due to the abstraction,

[13] `http://www.philipp.ruemmer.org/seneschal.shtml`

no linear ranking functions are found. On the original programs, the SAT-based approach and Seneschal find suitable ranking functions, on benchmark #34 however, the Model Checker times out afterwards.

Comparing the various techniques, we conclude that the simple SAT-based enumeration is most successful in synthesising useful ranking functions. It is able to prove 34 out of 51 terminating benchmarks and reports 27 as non-terminating. It does not time out on any instance.

Seneschal shows the second best performance: it proves 31 programs as terminating, almost as many as the SAT-based template approach. It reports 25 benchmarks as non-terminating and times out on 5.

For the experiments using Rankfinder[14], the bit-vector operators $+$, $\times$ with literals, $=$, $<_s$ and $<_u$ are approximated by the corresponding operations on the rationals, whereas nonexistence of ranking functions is reported for programs that use any other operations. Furthermore, we add constraints of the form $0 \leq v < 2^n$, where n is the bit-width of v, restricting the range of pre-state variables. This results in 23 successful termination proofs, and 35 cases of alleged non-termination. In three cases, the Model Checker times out on proving the final property, and in 5 cases Rankfinder returns an unsuitable ranking function.

For the two QBF techniques we used an experimental version of QuBE, which performed better than sKizzo, Quantor, and Squolem. The constrained template ($QBF[-1,+1]$) is still able to synthesise some useful ranking functions within the time limit. It proves 9 benchmarks terminating and reports 11 as non-terminating. The unconstrained approach (QBF $P(c,x)$), however, proves only 5 programs terminating and one non-terminating, with the QBF-Solver timing out on all other benchmarks.

We also implemented the approach suggested by Biere et al. [5] (bottom row of Table 2), which does not require ranking functions, but instead proves that an entry state of the loop is never revisited. Generally, these assertions are difficult for SATABS. While this method is able to show only 14 programs terminating, there are 4 benchmarks (#31, #45, #50, and #61) that none of the other methods can handle as they require non-linear ranking functions.

Our benchmark suite, all results with added detail, and additional experiments are available online at `http://www.cprover.org/termination/`.

5 Related work

Numerous efficient methods are now available for the purpose of finding ranking functions (e.g., [1, 6, 10, 14]). Some tools are complete for the class of ranking functions for which they are designed (e.g., [14]), others employ a set of heuristics (e.g., [1]). Until now, no known tool supported machine-level integers.

Bradley et al. [6] give a complete search-based algorithm to generate linear ranking functions together with supporting invariants for programs defined in Presburger arithmetic. We propose a related constraint-based method to synthesise linear ranking functions for such programs. It is worth noting that our

[14] `http://www.mpi-inf.mpg.de/~rybal/rankfinder/`

method is a decision procedure for the existence of linear ranking functions in this setting, while the procedure in [6] is sound and complete, but might not terminate when applied to programs that lack linear ranking functions. An experimental comparison with Bradley et al.'s method is future work.

Ranking function synthesis is not required if the program is purely a finite-state system. In particular, Biere, Artho and Schuppan describe a reduction of liveness properties to safety by means of a monitor construction [5]. The resulting safety checks require a comparison of the entire state vector whereas the safety checks for ranking functions refer only to few variables. Our experimental results indicate that the safety checks for ranking functions are in most cases easier. Another approach for proving termination of large finite-state systems was proposed by Ball et al. [2]; however, we would need to develop a technique to find suitable abstractions. Furthermore, since neither one of these techniques leads to ranking functions, it is not clear how they can be integrated into systems whose aim is to prove termination of programs that mix machine integers with data-structures, recursion, and/or numerical libraries with arbitrary precision.

6 Conclusion

The development of efficient ranking function synthesis tools has led to more powerful automatic program termination provers. While synthesis methods are available for a number of domains, efficient procedures for programs over machine integers have until now not been known. We have presented two new algorithms solving the problem of ranking function synthesis for bit-vectors: (i) a complete method based on a reduction to quantifier-free Presburger arithmetic, and (ii) a template-matching method for finding ranking functions of specified classes. Through experimentation with examples drawn from Windows device drivers we have shown their efficiency and applicability to systems-level code. The bottleneck of the methods is the reachability analysis engine. We will therefore consider optimizations for this engine specific to termination analysis as future work.

Acknowledgements. We would like to thank M. Narizzano for providing us with an experimental version of the QuBE QBF-Solver that outputs an assignment for the top-level existentials and H. Samulowitz for discussions about QBF encodings of the termination problem and for evaluating several QBF solvers. Besides, we are grateful for useful comments from Vijay D'Silva, Georg Weissenbacher, and the anonymous referees.

References

1. Babic, D., Hu, A.J., Rakamaric, Z., Cook, B.: Proving termination by divergence. In: SEFM. pp. 93–102. IEEE (2007)
2. Ball, T., Kupferman, O., Sagiv, M.: Leaping loops in the presence of abstraction. In: CAV. LNCS, vol. 4590, pp. 491–503. Springer (2007)

3. Benedetti, M.: sKizzo: A suite to evaluate and certify QBFs. In: CADE. LNCS, vol. 3632, pp. 369–376. Springer (2005)
4. Biere, A.: Resolve and expand. In: SAT. LNCS, vol. 3542, pp. 59–70. Springer (2005)
5. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: FMICS. ENTCS, vol. 66, pp. 160–177. Elsevier (2002)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination analysis of integer linear loops. In: CONCUR. LNCS, vol. 3653, pp. 488–502. Springer (2005)
7. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. FMSD 25(2-3), 105–127 (2004)
8. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: SAS. LNCS, vol. 3672, pp. 87–101. Springer (2005)
9. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI. pp. 415–426. ACM (2006)
10. Encrenaz, E., Finkel, A.: Automatic verification of counter systems with ranking functions. In: INFINITY. ENTCS, vol. 239, pp. 85–103. Elsevier (2009)
11. Giunchiglia, E., Narizzano, M., Tacchella, A.: QuBE++: an efficient QBF solver. In: FMCAD. LNCS, vol. 3312, pp. 201–213 (2004)
12. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI. pp. 35–46. ACM (1988)
13. Jussila, T., Biere, A., Sinz, C., Kroening, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: SAT. LNCS, vol. 4501, pp. 201–214. Springer (2007)
14. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI. LNCS, vol. 2937, pp. 239–251. Springer (2004)
15. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS. pp. 32–41. IEEE (2004)
16. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: PADL. LNCS, vol. 4354, pp. 245–259. Springer (2007)
17. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Sprawozdanie z I Kongresu metematyków slowiańskich, Warsaw 1929. pp. 92–101 (1930)
18. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: LPAR. LNCS, vol. 5330, pp. 274–289. Springer (2008)
19. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
20. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: CAV. LNCS, vol. 5123, pp. 385–398. Springer (2008)