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. [1].

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.

```
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).

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 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 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 width is statically declared for each variable and does not change during program execution. The syntactic categories of programs, statements, and expressions are defined by the following grammar:

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

Expressions $0_n, 1_n, \ldots$ are bit-vector literals of width n, $*_n$ non-deterministically 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

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

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

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. (For the complete typing rules see Appendix A.)

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

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

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

Lemma 1. *Deciding termination of bit-vector programs is PSPACE-complete in the sum of the program size and the size of the program's available memory (the sum of the bit-widths of the declared variables).*

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:*

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

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:

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

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

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 [2], 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* [3,4]. In this approach, termination of a program is first expressed as a *safety* property [1], 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 is an example of how to reach the loop, 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. This path constitutes a new, smaller, and loop-free program, which does not contain choice operators. Trivially, π is equivalent to a relation $R_\pi(x, x')$, where x and x' are states of the program. 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 three new methods to synthesise ranking functions for given paths in a bit-vector program: one based on integer linear programming techniques, and two methods that are based on propositional reasoning using SAT- and QBF-solvers. 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 [4] and ARMC [5] was developed by Podelski et al. [6]. 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 [7] is appropriate (Lem. 5 in the appendix). 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 [6]). *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

We extend the linear programming approach from Sect. 3.1 to fully support bit-vector programs. To this end, we first generalise Theorem 1 to disjunctions of systems of inequalities over the integers. We then define a complete 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:

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

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. 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 gives a complete criterion for *integral* systems, however: a system $Ax + A'x' \leq b$ 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), see Lem. 6 in Appendix C. Each 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 [7].

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:*

$$\lambda_1^i A'_i = 0, \quad \lambda_2^i (A_i + A'_i) = 0, \quad \lambda_2^i b_i < 0, \quad (\lambda_1^i - \lambda_2^i) A_i = 0, \quad \lambda_2^i A'_i = r. \quad (3)$$

A proof and an algorithm for strengthening transition disjuncts and applying this lemma is given in Appendix C.

Ranking functions for Presburger arithmetic. Presburger arithmetic (PA) is the first-order theory of integer arithmetic without multiplication [8]. We describe a complete procedure to generate linear ranking functions for PA-defined transition relations by reduction to Lem. 3. 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.

PA allows quantifier elimination by means of introducing divisibility constraints, which implies that we can always represent PA-defined transition relations in the form

$$R(x,x') \ \equiv \ \bigvee_{i=1}^{l} A_i x + A'_i x' \leq b_i \wedge D_i \quad (4)$$

where each D_i is a conjunction of divisibility constraints $\epsilon \mid (cx + dx')$. Divisibility constraints have to be introduced in certain situations during quantifier elimination, and state that the value of the term $cx + dx'$ is a multiple of the natural number $\epsilon \in \mathbb{N}^+$. To obtain form (4), it is necessary to rewrite equations $s = t$ as conjunctions $s \leq t \wedge t \leq s$ of inequalities, and to turn negated divisibility constraints $\epsilon \nmid t$ into disjunctions $(\epsilon \mid t+1) \vee (\epsilon \mid t+2) \vee \cdots \vee (\epsilon \mid t + \epsilon - 1)$.

Divisibility constraints have to be eliminated in order to apply Lem. 3, which is possible in two ways:

(i) The inequalities of a disjunct may imply lower and upper bounds on the value of $cx + dx'$ in a constraint $\epsilon \mid (cx + dx')$ in D_i, i.e., for some $a, b \in \mathbb{Z}$ it is the case that $A_i x + A'_i x' \leq b_i$ implies $a \leq cx + dx' \leq b$. Under this assumption, $\epsilon \mid (cx + dx')$ can equivalently be written as a finite disjunction:

$$\bigvee_{i = \lceil \frac{a}{\epsilon} \rceil, \ldots, \lfloor \frac{b}{\epsilon} \rfloor} i = cx + dx'$$

In particular, this is always the case for transition relations of bit-vector programs, because all variables range over finite domains.

(ii) In general, fresh variables can be used to turn divisibility constraints into equations (this can also lead to fewer considered cases than (i)). Given a constraint $\epsilon \mid (cx + dx')$, we introduce new pre-state variables y_c, y_d and post-state variables y'_c, y'_d, which are defined by adding conjuncts to $R(x, x')$:

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

Intuitively, y_c represents the expression $\lfloor \frac{cx}{\epsilon} \rfloor$, and y_d the expression $\lfloor \frac{dx}{\epsilon} \rfloor$. With the new variables, $\epsilon \mid (cx + dx')$ can be rewritten to the equivalent constraint $\epsilon \mid (cx - \epsilon y_c + dx' - \epsilon y'_d)$, which can then be replaced with a disjunction of equations as described in (i).

Iterating these steps eventually leads to a transition relation $R'(x, y, x', y')$ without divisibility judgments, such that $\exists y, y'. R'(x, y, x', y') \equiv R(x, x')$ (the vectors y, y' contain the variables introduced in step (ii)). We can then apply Lem. 3 to R' to derive a linear ranking function $m'(x, y)$. New variables y_c from (ii) that might occur in $m'(x, y)$ can now be interpreted as a division expression $\lfloor \frac{cx}{\epsilon} \rfloor$, so that $m'(x, y)$ can equivalently be written as a ranking function $m(x)$ (but not always as a *linear* function). If no division expressions are supposed to occur in $m(x)$, it is also possible to add further constraints to Eq. (3).

Lemma 4. *The procedure described above decides the existence of linear ranking functions for transition relations $R(x, x')$ defined in Presburger arithmetic.*

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:

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

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 course.

Example We consider the bit-vector program (1) corresponding to Fig. 2. An encoding of the program in PA is:

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

From the side conditions, we can derive that the term $nLoop' - nLoop - 2^{32} + 4$ has the range $[5 - 2^{33}, 3]$, so that the divisibility constraint can be split into two cases. With some further simplifications, we obtain:

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

It is now easy to see that each disjunct is satisfiable and integral, which means that Lem. 3 is applicable. Because the conditions (3) 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 $x = (x_1, \dots, x_{|\mathcal{X}|})$ be a vector of program variables and associate a coefficient c_i with each $x_i \in \mathcal{X}$. The coefficients constitute the vector $c = (c_1, \dots, c_{|\mathcal{X}|})$. 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*

$$\exists c\ \forall x, x' \,.\, R_\pi(x, x') \Rightarrow p(c, x') <_s p(c, x)\ . \tag{5}$$

Vice versa, if there exists a linear ranking function for π, then Eq. (5) must be valid for $n = \max_i(\alpha(x_i))$ and

$$w \geq n \cdot (|\mathcal{X}| - 1) + |\mathcal{X}| \cdot \log_2 |\mathcal{X}| + 1\ .$$

For a proof of this theorem, see Appendix D.

It is straightforward to flatten Eq. (5) into QBF. Thus, a QBF solver that returns an assignment for the top-level existential variables can compute suitable coefficients. Examples of such solvers are Quantor [9], sKizzo [10], and Squolem [11]. In our experiments, we use an experimental version of QuBE [12].

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 5 into

$$\neg \exists x, x' \; . \; R_\pi(x, x') \wedge \neg(p(\gamma(c), x') <_s p(\gamma(c), x)) \; , \tag{6}$$

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 p is *not* a genuine ranking function. Thus, we enumerate all possible γ until we find one for which Equation 6 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 thus obtain 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 6 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. [4], we implemented a binary reachability analysis engine to evaluate our ranking synthesis methods. Our implementation uses SATABS as the reachability checker [13], which implements SAT-based predicate abstraction. Our benchmarks are device drivers from the Windows Driver Development Kit (WDK).[6] The WDK already includes verification harnesses for the drivers. We use GOTO-CC[7] 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. [16]).

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

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

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.

Slicing the input. Just like Cook et al. [4], 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 [17] of the program, slicing backwards from the termination assertion.[8] With the 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.[9] 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.[10]

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 the loop, the problem would persist in a concurrent setting, where a device may be disconnected while the loop is executed. An excerpt of the source code of the driver is in Appendix F.

[8] We remove all instructions that the loop does not (syntactically) depend on, while keeping the control flow graph intact. 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).

[9] `http://www.cprover.org/satabs/termination/`

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

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 removing all source code that does not affect program termination (much like an automated slicer, but more thoroughly). 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.

#	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. [1]	–	–	–	●	–	–	–	–	–	○	–	●	●	–	–	–	–	–	–	●	–	●	●	–	–	–	–	○	–	–	●

#	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	L	N	L	L	L	L	L	N	T	
SAT	○	●	–	○	●	○	●	●	●	●	●	●	○	○	●	●	○	○	○	●	○	○	○	●	●	●	●	●	○	○	
Seneschal	○	●	–	–	●	○	●	–	●	●	●	●	–	○	●	●	○	○	○	●	○	–	○	●	●	●	–	–	○	–	
Rankfinder	○	◐	–	○	●	◐	●	●	○	○	●	●	○	○	●	◐	○	○	○	●	◐	○	–	●	●	●	◐	○	○	○	
QBF [-1,+1]	–	–	–	–	–	–	–	–	●	●	–	●	○	–	–	●	–	○	○	–	○	–	–	●	–	–	–	●	○	–	
QBF $P(c,x)$	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	●	–	–	
Biere et al. [1]	–	–	–	○	●	–	–	–	●	–	–	–	○	●	●	●	–	–	●	●	–	–	–	–	–	–	–	–	○	●	

● – Termination was proven
○ – (Possibly) Non-terminating
◐ – Incorrect under bit-vector semantics
– – Memory or time limits exhausted

T – Terminating (non-linear)
L – Terminating, and linear ranking functions exist.
N – Non-terminating

Table 2. Experimental Data

To evaluate the integer linear programming method described in Sect. 3.2, we developed the prototype Seneschal.[11] 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.

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

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

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.

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 35 out of 51 terminating benchmarks and reports 25 as non-terminating. It times out on only a single instance, after reaching a suitable template instantiation, because the SAT solver requires too much time to determine unsatisfiability.

Seneschal shows the second best performance: it proves 27 programs as terminating, almost as many as the SAT-based template approach. It reports 18 benchmarks as non-terminating and times out on 16.

For the experiments using Rankfinder[12], 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 added constraints restricting the range of pre-state variables to the input, i.e., constraints of the form $0 \leq v < 2^n$, where n is the bit-width of v. The ranking functions returned by Rankfinder do not contain any information about the bit-widths to be used, and we therefore use a typing that avoids overflows. This results in 18 successful termination proofs, and 28 cases of alleged non-termination. In three cases, the Model Checker times out on proving the final property, and in 12 cases Rankfinder returns an unsuitable ranking function. The runtimes of the SAT coefficient enumeration, Seneschal, and Rankfinder are relatively small (more detail is in Appendix G).

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. The unconstrained approach (QBF $P(C, X)$), however, proves only a single program terminating, with the QBF-Solver timing out on all other benchmarks.

We also implemented the approach suggested by Biere et al. [1] (rightmost column of Table 2), which does not require a ranking function, but instead proves an assertion that an entry state of the loop is never reached again. Generally, these assertions are difficult for SATABS. While this method is able to show only 14 programs terminating, there are four 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/satabs/termination/`.

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

5 Related work

Numerous efficient methods are now available for the purpose of finding ranking functions (e.g., [6, 19–21]). Some tools are complete for the class of ranking functions for which they are designed to search (e.g., [6]), others employ a set of heuristics (e.g., [21]). Until now, no known tool supported machine-level integers.

Bradley et al. [19] 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 method is a decision procedure for the existence of linear ranking functions in this setting, while the procedure in [19] 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 [1]. 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. [22]; 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-level 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.

References

1. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: FMICS. Number 66 in ENTCS, Elsevier (2002)
2. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, IEEE (2004) 32–41
3. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: SAS. Number 3672 in LNCS, Springer (2005) 87–101
4. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, ACM (2006) 415–426
5. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: PADL. Number 4354 in LNCS, Springer (2007) 245–259
6. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI. Number 2937 in LNCS, Springer (2004) 239–251
7. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
8. 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, Warszawa 1929, Warsaw, Poland (1930) 92–101,395
9. Biere, A.: Resolve and expand. In: SAT. Number 3542 in LNCS, Springer (2005) 59–70
10. Benedetti, M.: sKizzo: A suite to evaluate and certify QBFs. In: CADE. Number 3632 in LNCS, Springer (2005) 369–376
11. Jussila, T., Biere, A., Sinz, C., Kroening, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: SAT. Number 4501 in LNCS, Springer (2007) 201–214
12. Giunchiglia, E., Narizzano, M., Tacchella, A.: QuBE++: an efficient QBF solver. In: FMCAD. Number 3312 in LNCS (2004) 201–213
13. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. FMSD **25** (2004) 105–127
14. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys, ACM (2006) 73–85
15. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: CAV. Volume 2102 of LNCS., Springer (2001) 260–264
16. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: CAV. Volume 5123 of LNCS., Springer (2008) 385–398
17. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI, ACM (1988) 35–46
18. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: LPAR. Number 5330 in LNCS, Springer (2008) 274–289
19. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination analysis of integer linear loops. In: CONCUR. Volume 3653 of LNCS., Springer (2005) 488–502
20. Encrenaz, E., Finkel, A.: Automatic verification of counter systems with ranking functions. In: INFINITY. ENTCS, Elsevier (2009) To appear.
21. Babic, D., Hu, A.J., Rakamaric, Z., Cook, B.: Proving termination by divergence. In: SEFM, IEEE Computer Society (2007) 93–102
22. Ball, T., Kupferman, O., Sagiv, M.: Leaping loops in the presence of abstraction. In: CAV. Number 4590 in LNCS, Springer (2007) 491–503

A Typing Rules for Bit-Vector Programs

We write $t : n$ to denote that the expression t is correctly typed and denotes a bit-vector of length n. Given a statement or program β, we write $\beta : \bot$ to express that β is correctly typed. In the following rules, $x \in \mathcal{X}$ ranges over variables, $n \in \mathbb{N}^+$ over positive natural numbers, s, t over expressions, β, γ over statements:

$$\frac{k \in \mathbb{N}^+}{k_n : n} \qquad \frac{}{*_n : n} \qquad \frac{t : n}{\neg t : n} \qquad \frac{s : n \quad t : n}{s \circ t : n}\ \circ \in \{+, \times, \div, \ \&\ , \mid\}$$

$$\frac{\alpha(x) = n}{x : n} \qquad \frac{s : n \quad t : k}{s \circ t : n}\ \circ \in \{\ll, \gg\} \qquad \frac{\alpha(x) = n \quad t : n}{x := t : \bot}$$

$$\frac{t : k}{\mathsf{cast}_n(t) : n} \qquad \frac{s : n \quad t : n}{s \circ t : 1}\ \circ \in \{=, \leq\} \qquad \frac{t : 1}{\mathsf{assume}\ (t) : \bot}$$

$$\frac{}{\mathsf{skip} : \bot} \qquad \frac{\beta : \bot \quad \gamma : \bot}{\beta; \gamma : \bot} \qquad \frac{\beta : \bot \quad \gamma : \bot}{\beta \,\square\, \gamma : \bot} \qquad \frac{\beta : \bot \quad \gamma : \bot}{\beta\ \mathsf{repeat}\ \{\ \gamma\ \} : \bot}$$

B Termination of Bit-Vector Programs is PSPACE-complete

Bit-vector programs (as defined in the previous section, i.e., not providing heap or recursion) belong to the class of *constant memory* programs, which means that the memory consumption is defined upfront and does not depend on program inputs. The termination of such programs is decidable, more precisely, it is PSPACE-complete: polynomial memory is needed in the size of the program and the size of the program's available memory.

Termination is PSPACE-hard. We first show that the termination problem for bit-vector programs is PSPACE-hard by polynomial reduction of the satisfiability problem of QBF-formulae (which is the canonical PSPACE-complete formalism [1]). Suppose $\phi = Q_1 x_1. \cdots Q_n x_n. \psi$ is a closed QBF-formula in prenex form, where $Q_i \in \{\forall, \exists\}$ and ψ is quantifier-free. We will write a program of polynomial size and memory consumption (in the size of ϕ) that terminates if and only if ϕ is satisfiable. To this end, we assume that $x_1, \ldots, x_n$ are also declared as program variables of bit-width 1, i.e., $\alpha(x_i) = 1$ for $i \in \{1, \ldots, n\}$. Furthermore, we assume that the matrix ψ is an expression in the grammar defined in Table **??**, which is no restriction because the language provides the boolean operators $\&$, $\mid$, $\neg$.

We need further variables to write the program to check satisfiability of ϕ: a variable *num* with $\alpha(num) = \lceil \log_2(n+2) \rceil$, variables $r_1, \ldots, r_n$ with $\alpha(r_i) = 1$, and variables $num_1, \ldots, num_n$ with $\alpha(num_i) = 2$.

The satisfiability checker has the following form (for sake of brevity, we omit the bit-widths of literals like 5_3):

```
$num := 1;\ num_1 := 0;\ \cdots;\ num_n := 0$
    repeat {    assume $(num = 0\ \&\ \neg r_1)$
                $\Box\ loop_1\ \Box\ loop_2\ \Box\ \cdots\ \Box\ loop_n$
                $\Box\ \big($assume $(num = n+1);\ r_{n+1} := \phi;\ num := n\big)$ }
```

Each of the blocks $loop_i$ is responsible for enumerating the possible values of x_i and evaluating the quantifier Q_i:

```
assume $(num = i)$;
  (    (assume $(num_i = 0);\ num_i := 1;\ num := i+1;\ x_i := 0)$
    $\Box$ (assume $(num_i = 1);\ num_i := 2;\ num := i+1;\ r_i := r_{i+1};\ x_i := 1)$
    $\Box$ (assume $(num_i = 2);\ num_i := 0;\ num := i-1;\ r_i := r_i \circ r_{i+1})$    )
```

where $\circ =\ \&$ for $Q_i = \forall$, and $\circ =\ |$ for $Q_i = \exists$.

We can observe that the size of each $loop_i$ is logarithmic in n (because numbers in the range $0, \ldots, n+1$ need to be encoded). The size of all $loop_i$ blocks together is therefore in $O(n \log n)$, and the size of the whole satisfiability checker is in $O(s \log s)$, where s is the size of the formula ϕ.

The memory consumption of the satisfiability checker is

n	variables x_i
$+ \lceil \log_2(n+2) \rceil$	variable num
$+ n$	variables r_i
$+ 2n$	variables num_i
$= 4n + \lceil \log_2(n+2) \rceil \in O(n)$	

Finally, it can be observed that the transformation of QBF-formulae into prenex form, as well as the generation of the satisfiability checker can be done in polynomial time.

Termination is in PSPACE. To prove that the termination of bit-vector programs is in PSPACE, we encode the termination of a program α into a QBF-formula of polynomial size in the size of α and α's available memory. Because the satisfiability of QBF-formulae is in PSPACE [1], this shows that program termination is in PSPACE as well (and therefore, together with the previous paragraphs, termination is PSPACE-complete). The construction is based on the classical proof that QBF is PSPACE-complete [1] and uses a technique called "squaring abbreviation."

We first assume that the transition relations R_β, R_γ of a bit-vector program β repeat $\{ \gamma \}$ are encoded as quantifier-free boolean formulae $\phi_\beta(x, x')$ and $\phi_\gamma(x, x')$ (note, that the encoding can be chosen such that the size of $\phi_\beta(x, x')$

and $\phi_\gamma(x, x')$ is polynomial in the size of β, γ). We then recursively define a predicate $\mathit{reach}(a, b, n)$ with the intended semantics "the statement γ can reach the state b from state a in at most 2^n steps." A naive recursive definition of $\mathit{reach}(a, b, n)$ is:

$$\begin{aligned}\mathit{reach}(a, b, 0) &\equiv a = b \vee \phi_\gamma(a, b)\\ \mathit{reach}(a, b, n) &\equiv \exists c. \mathit{reach}(a, c, n-1) \wedge \mathit{reach}(c, b, n-1)\end{aligned}$$

Expanding $\mathit{reach}(a, b, n)$ in this way will obviously lead to a formula that is exponential in size, but that only contains existential quantifiers.

Alternatively, we can choose the definition:

$$\begin{aligned}\mathit{reach}(a, b, 0) &\equiv a = b \vee \phi_\gamma(a, b)\\ \mathit{reach}(a, b, n) &\equiv \exists c. \forall a', b'. \begin{pmatrix} a' = a \wedge b' = c \vee a' = c \wedge b' = b \\ \rightarrow \mathit{reach}(a', b', n-1) \end{pmatrix}\end{aligned}$$

Because there is no right-hand side with more than one occurrence of *reach*, this leads to a QBF-formula whose size is polynomial in n and the size of γ defining $\mathit{reach}(a, b, n)$.

The predicate *reach* can now be used to encode termination as a QBF-formula: due to the finiteness of the state space, it is sufficient to construct a formula that states the absence of *lassos* in the transition graph. Assuming that the state space has 2^n elements (i.e., n is the sum of the bit-widths of the variables declared in the program), this formula is:

$$\neg \exists a, b, c, d. \ (\phi_\beta(a, b) \wedge \mathit{reach}(b, c, n) \wedge \phi_\gamma(c, d) \wedge \mathit{reach}(d, c, n))$$

Altogether, the size of the formula is polynomial in n and the size of β, γ, and the formula can obviously be generated from β, γ in polynomial time.

Note that this encoding is equivalent to expressing the termination property as a safety property (e.g, according to [2]), and subsequent application of the QBF-based Bounded Model Checking technique introduced in [3].

C Ranking Functions for Disjunctive Integer Transition Relations

The mathematical result underlying both Podelski and Rybalchenko's method [4] and our method described in Sect. 3.2 is Farkas' lemma [5]:

Lemma 5 (Farkas' lemma). *Suppose $A \in \mathbb{Q}^{n \times k}$ is a matrix, $b \in \mathbb{Q}^n$ a vector such that the system $Ax \le b$ of inequalities is satisfiable, $c \in \mathbb{Q}^k$ is a (row) vector, and $\delta \in \mathbb{Q}$ is a rational. Then*

$$\{x \in \mathbb{Q}^k : Ax \le b\} \subseteq \{x \in \mathbb{Q}^k : cx \le \delta\} \tag{7}$$

if and only if there is a non-negative (row) vector $\gamma \in \mathbb{Q}^n$ such that $\gamma A = c$ and $\gamma b \le \delta$.

The equivalence does *not* hold if x in (7) ranges over the integers; implied inequalities in this case can in general not be represented as non-negative linear combinations. Farkas' lemma still works, however, in the special case of *integral* systems of inequalities. A system $Ax \leq b$ is called integral if the polyhedron $\{x \in \mathbb{Q}^n \mid Ax \leq b\}$ coincides with its integral hull (the convex hull of the integer points contained in it).[13]

Lemma 6 (Integral version of Farkas' lemma). *Suppose $A \in \mathbb{Q}^{n\times k}$ is a matrix, $b \in \mathbb{Q}^n$ a vector such that the system $Ax \leq b$ of inequalities is satisfiable* and integral, *$c \in \mathbb{Q}^k$ is a (row) vector, and $\delta \in \mathbb{Q}$ is a rational. Then*

$$\{x \in \mathbb{Z}^k : Ax \leq b\} \subseteq \{x \in \mathbb{Z}^k : cx \leq \delta\} \tag{8}$$

if and only if there is a non-negative (row) vector $\gamma \in \mathbb{Q}^n$ such that $\gamma A = c$ and $\gamma b \leq \delta$.

Proof. We show that (7) if and only if (8) in the case of an integral system $Ax \leq b$. The conjecture then follows by Lem. 5.

(7) $\Rightarrow$ (8): holds because of $\mathbb{Z} \subset \mathbb{Q}$.

(8) $\Rightarrow$ (7): suppose (8) holds. This implies that the convex hull of the set $\{x \in \mathbb{Z}^k : Ax \leq b\}$ is contained in the half-space $\{x \in \mathbb{Q}^k : cx \leq \delta\}$. The convex hull of $\{x \in \mathbb{Z}^k : Ax \leq b\}$ is the same as the integral hull of $\{x \in \mathbb{Q}^k : Ax \leq b\}$, which coincides with $\{x \in \mathbb{Q}^k : Ax \leq b\}$ because $Ax \leq b$ is integral. Therefore (7).

Derivation of integral systems One approach to transform an arbitrary system $Ax \leq b$ of inequalities into an integral system with the same integer solutions is as follows: first, we derive an equivalent *total dual integral* system $A'x \leq b'$ from $Ax \leq b$ such that $A' \in \mathbb{Z}^{n' \times k}$. A system $A'x \leq b'$ is total dual integral if the duality equation

$$\max\{x \in \mathbb{Q}^k : A'x \leq b'\} = \min\{yb : y \geq 0,\, yA' = c\}$$

has an integral optimum solution y for each integral vector c for which the minimum is finite [5]. $A'x \leq b'$ can then be strengthened to $A'x \leq \lfloor b' \rfloor$ without losing integer solutions. Iterating this procedure eventually leads to an integral system of inequalities.

Linear ranking functions over the integers We are now in the position to give a proof for Lem. 3:

Proof. 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. We show the two directions of the equivalence:

[13] This deviates from the terminology in [5], where integrality is attributed to polyhedra, and not to systems of inequalities. We choose to speak of integral systems of inequalities for sake of brevity.

```
Input: Transition relation $R(x, x') \equiv \bigvee_{i=1}^{l} A_i x + A'_i x' \le b_i$
Output: Ranking function $m(x) = rx - c$, or answer "No linear ranking
        function exists"
1  foreach integral disjunct $A_i x + A'_i x' \le b_i$ do
2      if $A_i x + A'_i x' \le b_i$ has no ranking function according to Theorem 1 then
3          return No linear ranking function exists;
4  while true do
5      if there are non-negative vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^k$ such that (3) holds then
6          $c \leftarrow \min\{rx' : \exists x. R(x, x')\}$;
7          return $m(x) = rx - c$ is a ranking function;
8      if all disjuncts $A_i x + A'_i x' \le b_i$ are integral then
9          return No linear ranking function exists;
10     Pick a non-integral disjunct $A_i x + A'_i x' \le b_i$;
11     Strengthen $A_i x + A'_i x' \le b_i$ as in Sect. C;
12     if $A_i x + A'_i x' \le b_i$ is integral and has no ranking function according to
       Theorem 1 then
13         return No linear ranking function exists;
```

Algorithm 1: Ranking functions for disjunctive integer transition relations with iterative strengthening

$\Rightarrow$: Assume the relation $R(x, x')$ has a linear ranking function $m(x) = rx + c$. Arguing as in the proof [4, Theorem 2], this means that for some $\delta > 0$ and all $i \in \{1, \dots, l\}$ we have:

$$\text{for all } x, x' \in \mathbb{Z}^n : A_i x + A'_i x' \le b_i \text{ implies} \\ rx + c \ge 0 \wedge rx' + c \ge 0 \wedge rx' + \delta \le rx \quad (9)$$

By Lem. 6, this implies that there are non-negative vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^k$ such that for $i \in \{1, \dots, l\}$:

$$\begin{array}{lll} \lambda_1^i A_i = -r, & \lambda_1^i A'_i = 0, & \lambda_1^i b_i \le c, \\ \lambda_2^i A_i = -r, & \lambda_2^i A'_i = r, & \lambda_2^i b_i \le -\delta \end{array}$$

It is now easy to see that (3) is implied by these equations and inequalities.

$\Leftarrow$: Assume (3) holds for non-negative vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^k$ for $i \in \{1, \dots, l\}$. By Theorem 1, for each $i \in \{1, \dots, l\}$ the disjunct $A_i x + A'_i x' \le b_i$ has a linear ranking function of the form $m_i(x) = rx + c_i$, which implies that

$$m(x) = rx + \min\{c_i : i \in \{1, \dots, l\}\}$$

is a ranking function for $R(x, x')$.

Linear ranking functions for programs in Presburger arithmetic We give a proof for Lem. 4:

Proof. Suppose $R(x, x')$ is a formula in Presburger arithmetic. By eliminating quantifiers, negated equations, negated divisibility constraints, and divisibility constraints in R by means of new variables, we derive a transition relation $R'(x, y, x', y')$ with:

$$\begin{aligned} R'(x, y, x', y') &\equiv \bigvee_{i=1}^{l} A_i \begin{pmatrix} x \\ y \end{pmatrix} + A'_i \begin{pmatrix} x' \\ y' \end{pmatrix} \leq b_i \\ \exists y, y'. R'(x, y, x', y') &\equiv R(x, x') \end{aligned} \tag{10}$$

We can assume that every disjunct $A_i\binom{x}{y} + A'_i\binom{x'}{y'} \leq b_i$ is satisfiable and integral, because unsatisfiable disjuncts can be detected and left out, and because non-integral disjuncts can be transformed into integral disjuncts.

Because of (10), every linear ranking function of $R(x, x')$ is also a ranking function of $R'(x, y, x', y')$. This means that we can apply Lem. 3 to $R'(x, y, x', y')$ and simply impose the additional constraints

$$r_{n+1} = r_{n+2} = \cdots = 0$$

where $r = (r_1, r_2, \ldots)^t$ is the coefficient vector to be determined in Lem. 3, and $n = |x|$ is the length of the x and x' vector. This ensures that the resulting ranking functions $m'(x, y) = r\binom{x}{y} + c = m(x)$ only depends on x and not on y.

D Synthesis of Ranking Functions from Templates

We investigate the maximal size of template coefficients needed in Sect. 3.3 in order to obtain a complete procedure, i.e., the number of bits to be reserved as stated in Theorem 2.

A lower bound To show that our upper bound is reasonably tight, we first give a lower bound on the number of bits required. Consider terminating programs (for which linear rank functions exist) of the following form:

```
i=1;
while $i \neq 0 \vee j \neq 0 \vee k \neq 0 \ldots$ do
    ...
    $k := k + (i \div 255) \times (j \div 255)$;
    $j := j + i \div 255$;
    $i := i + 1$;
```

where the width of all variables is n. Programs built after this scheme require ranking functions that order variables lexicographically. A suitable ranking function is of the form

$$\cdots + c_k \times k + c_j \times j + c_i \times i\ ,$$

where $c_i = -1$, $c_j = -2^n$, $c_k = -2^{2n}$, etc. This means that the corresponding bit-widths of the coefficients are $\alpha(c_i) = 2$, $\alpha(c_j) = n + 2$, and $\alpha(c_k) = 2n + 2$. Note that the constant 2 arises from the fact that each coefficient is of the form

-2^x for some x, which is not contained in $[0, 2^x)$, and we thus need an extra bit to represent a coefficient of this size and its sign.

The total number of bits required to represent all coefficients is thus

$$\sum_{i=0}^{|\mathcal{X}|-1} i \cdot n + 2 = 2 \cdot |\mathcal{X}| + n \cdot \sum_{i=0}^{|\mathcal{X}|-1} i \ ,$$

which we can also write as

$$2 \cdot |\mathcal{X}| + n \cdot \frac{|\mathcal{X}| \cdot (|\mathcal{X}| - 1)}{2} \ .$$

I.e., we require $O(|\mathcal{X}|^2 \cdot n)$ bits.

An upper bound In the following, we consider a transition relation $R(s, s')$ over a vocabulary $\mathcal{X}$ of variables. For sake of simplicity, it is assume that the bit-width of all $|\mathcal{X}| = m$ variables is n, i.e., $\alpha(x) = n$ for all $x \in \mathcal{X}$ (this means that all variables range over $[0, 2^n)$). States $s \in \mathcal{S}$ are identified with elements of the vector space $\mathbb{Q}^{|\mathcal{X}|}$, for an arbitrary but fixed enumeration $x_1, \ldots, x_m$ of the variables $\mathcal{X}$. The candidates for ranking functions are the elements of the vector space $V = \mathbb{Q}^{|\mathcal{X}|} \to \mathbb{Q}$ of rational linear functions over the program variables $\mathcal{X}$.

Every function $f \in V$ determines an order $\prec_f$ on the state space $\mathcal{S}$:

$$s \prec_f s' \ \equiv \ f(s) < f(s')$$

If $\prec_f$ is a (strict) total order, we say that f is *perfectly separating.* If two functions f, f' determine the same order, i.e., $\prec_f = \prec_{f'}$, we call them *equivalent.* Obviously, as $\mathcal{S}$ is finite, V is partitioned into finitely many equivalence classes in this way. We make the following observations:

– Because functions f, f' in the same equivalence class can prove the termination of exactly the same loops, it is sufficient to consider ranking function templates that represent at least one function from each equivalence class.
– The classes of perfectly separating functions subsume the classes of non-perfect functions, in the sense that every loop that can be proven terminating using the latter can also be proven terminating by perfectly separating functions.

When choosing the bit-width of coefficients in ranking function templates, it is therefore only necessary to ensure that the template represents at least one function in each class of perfectly separating functions.

The geometry of equivalence classes There is a simple geometric interpretation of equivalence classes. A non-perfect function f has the property that $f(s) = f(s')$ for some states $s \neq s'$. Because states are interpreted as vectors, this is equivalent to $f(s - s') = 0$. For any two states $s \neq s' \in \mathcal{S}$, we can observe that the set $E_{s,s'} = \{f \in V : f(s - s') = 0\}$ is a hyperplane of the vector space V, which altogether means that the set of non-perfect functions is the finite union

$$E \;=\; \bigcup_{s \neq s' \in \mathcal{S}} E_{s,s'}$$

of hyperplanes. The inverse $P = V \setminus E$ is then a finite union of open convex sets, each of which forms the interior of a convex (but unbounded) polyhedron.

These polyhedra coincide with the equivalence classes of perfectly separating functions. To see this, note that for each state $s \in \mathcal{S}$ the function $v_s : V \to \mathbb{Q}$, $v_s(f) = f(s)$ is continuous. This implies that if $f, f' \in V$ are perfectly separating functions that are not equivalent, every continuous path from f to f' in V has to cross E. Furthermore, the classes belonging to different polyhedra are distinct: for each hyperplane $E_{s,s'}$, it holds that $f(s - s') > 0$ for the functions f on one side of the hyperplane, and $f'(s - s') < 0$ for the functions f' on the other (because $f(s - s')$ is linear in f), which implies $s \prec_f s'$ and $s' \prec_{f'} s$.

Choosing representatives from equivalence classes To distinguish a basis of the vector space V, we first define states $s_1, \ldots, s_m \in \mathcal{S}$ by:

$$s_i(x_j) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

The basis $\{b_1, \ldots, b_m\} \subseteq V$ of V is then defined (as the dual basis) by:

$$b_i(s_j) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

This allows to represent functions in $f \in V$ in the form $\alpha_1 b_1 + \cdots + \alpha_m b_m$, which intuitively can be understood as

$$f(x_1, \ldots, x_m) \;=\; \alpha_1 x_1 + \cdots + \alpha_m x_m$$

In the sequel, we consider linear combinations with integer coefficients $\alpha_1, \ldots, \alpha_m$, and derive bounds on the absolute values of the coefficients such that still elements from each class of perfectly separating functions can be represented. These bounds determine the number of bits needed in ranking function templates.

We fix a non-empty class $A \subseteq V$ of perfectly separating functions, and assume that $E' \subseteq E$ is the union of all hyperplanes $E_{s,s'}$ that bound A. Each intersection of $m - 1$ such hyperplanes (provided that no two of them are parallel) is a straight line l that is adjacent to A. Now we can observe:

(i) Each such line l is generated by a function

$$f_l = \alpha_1 b_1 + \cdots + \alpha_m b_m$$

where $|\alpha_i| \leq 2^{n(m-1)} \cdot (m - 1)!$ for $i \in \{1, \ldots, m\}$.

(ii) The set A contains a function

$$f_A = \beta_1 b_1 + \cdots + \beta_m b_m$$

where $|\beta_i| \leq 2^{n(m-1)} \cdot m!$ for $i \in \{1, \ldots, m\}$.

To see that (i) holds, note that each hyperplane $E_{s,s'}$ is described by a linear equation $f(s - s') = 0$ that can be expanded to

$$\big(s(x_1) - s'(x_1)\big)\gamma_1 + \cdots + \big(s(x_m) - s'(x_m)\big)\gamma_m = 0$$

if the representation $f = \gamma_1 b_1 + \cdots + \gamma_m b_m$ is chosen. Coefficients $s(x_1) - s'(x_1)$ in such an equation are in the range $[-2^n + 1, 2^n - 1]$. In order to find a vector in the intersection of $m - 1$ hyperplanes, we therefore need to solve a system of $m - 1$ linear equations:

$$\begin{array}{llll} v_1^1 \gamma_1 & + \ \cdots & + \ v_m^1 \gamma_m & = 0 \\ & & & \vdots \\ v_1^{m-1} \gamma_1 & + \ \cdots & + \ v_m^{m-1} \gamma_m & = 0 \end{array}$$

By elementary algebra, an integer solution to this system can be found by computing the following determinant (S_m is the group of permutations of $\{1, \ldots, m\}$, and the parity $\operatorname{sgn}(\sigma)$ is $+1$ for even and -1 for odd permutations σ):

$$\begin{vmatrix} v_1^1 & \ldots & v_m^1 \\ \cdots & \cdots & \cdots \\ v_1^{m-1} & \ldots & v_m^{m-1} \\ b_1 & \ldots & b_m \end{vmatrix} = \sum_{i=1}^{m} \sum_{\substack{\sigma \in S_m \\ \sigma(m)=i}} \operatorname{sgn}(\sigma) \Big(\prod_{j=1}^{m-1} v_{\sigma(j)}^j \Big) b_i$$

Because of $|v_i^j| < 2^n$ and $|S_m| = m!$, the absolute value of the coefficient of each basis vector b_i on the right-hand side is bounded by $2^{n(m-1)} \cdot (m-1)!$.

For (ii), we assume that there are m linearly independent lines $l_1, \ldots, l_m$ (as in (i)) that are adjacent to A. In this case, because A is convex, there is a sum

$$f_A = c_1 f_{l_1} + \cdots + c_m f_{l_m} \in A$$

with $c_i \in \{-1, +1\}$ for each $i \in \{1, \ldots, m\}$. The bounds on the absolute values of coefficients follow from (i). A similar argument can be used in the case that no m linearly independent lines exist.

From (i), we can derive the number of bits needed for Theorem 2:

$$\log_2(2^{n(m-1)} \cdot m!) \ = \ n(m-1) + \log_2 m! \ \leq \ n(m-1) + m \log_2 m$$

Because both positive and negative coefficients have to be represented, we need a further bit for the sign, which yields the bound $n(m-1) + m \log_2 m + 1$ given in Theorem 2.

E Bitwise Synthesis of Ranking Functions

Since ranking functions for bit-vector programs may naturally be expressed as bit-vector functions, i.e., $f : \mathbb{B}^\sigma \to \mathbb{B}^m$, where $\sigma = \sum_{x \in \mathcal{X}} \alpha(x)$ is the width of the program state, the question for the minimal required width m of the ranking function for a given R_π arises naturally. This gives rise to an incremental algorithm, which tries to synthesise ranking functions of increasing width $m = 1, 2, \ldots \sigma$, exploiting the fact that it is easier to find smaller functions.

Let s be a bit-vector of size σ, representing the program state. We can then think of bitwise templates for ranking functions of various types. For example, we define four function classes:

- Affine: $(c_1 \;\&\; s_1) \oplus (c_2 \;\&\; s_2) \oplus \cdots \oplus (c_\sigma \;\&\; s_\sigma) \oplus k$
- Conjunctive: $(c_1 \mid s_1) \;\&\; (c_2 \mid s_2) \;\&\; \cdots \;\&\; (c_\sigma \mid s_\sigma) \;\&\; k$
- Disjunctive: $(c_1 \;\&\; s_1) \mid (c_2 \;\&\; s_2) \mid \cdots \mid (c_\sigma \;\&\; s_\sigma) \mid k$
- Projective: $(c_1)?s_1 : (c_2)?s_2 : \cdots ?s_{\sigma-1} : s_\sigma$, where $a?b : c$ is the if-then-else operator.

This allows us to synthesise 1-bit ranking functions. For n-bit functions, we simply concatenate n templates, i.e., we use the tuple $(f_1(s), f_2(s), \ldots, f_n(s))$, where each f_i is a template of the same function class. We interpret the result as an element of $\mathbb{Z}_{2^n}$. The construction above enables the construction of 'narrow' ranking functions, but also allows us to choose a class of functions from Post's lattice of Boolean functions [6]. As an example, if we find (heuristically) that searching for $k+1$-bit projections is unlikely to be fruitful, we may increase the expressiveness of the template instead of increasing the bit-width (or vice versa).

Concatenated functions are not necessarily from the same function class as their components, and thus may allow us to synthesise ranking functions that we would not have found using linear templates. In particular, the concatenation $(f_1(s), f_2(s))$ of affine functions f_1, f_2 is not necessarily affine in $\mathbb{Z}_4$. In general, however, this approach allows us to generate *arbitrary* ranking functions, if we allow each f_i to be drawn from the top of Post's lattice, and the size of the concatenation is σ.

While the general idea of the algorithm is appealing, our implementation based on QBF-Solving (using templates as in Sec. 3.3) performs rather poorly in an experimental evaluation on our benchmarks: No useful results were obtained due to the QBF solver timing out on virtually all examples. On those where it does return a ranking function, the reachability checker times out subsequently.

F A practical termination problem

In module bulkpnp of the WDK sample USB driver (bulkusb) the driver requests an interface description structure to be searched within a ConfigurationDescriptor for every device available. It increments the loop counter if this did not return an error. The function USBD_ParseConfigurationDescriptorEx, however, is an API function for which no implementation is available. According to the API

```
while(iNumber < numberOfInterfaces) {
  iDesc = USBD_ParseConfigurationDescriptorEx(
                    ConfigurationDescriptor,
                    ConfigurationDescriptor,
                    iIndex,
                    0, -1, -1, -1);
  if(iDesc) {
    /* ... */
    iNumber++;
  }
  iIndex++;
}
```

Fig. 3. Code fragment from usb/bulkusb/sys/bulkpnp.c (simplified)

documentation, it may return NULL if no interface matches the search criteria (`iIndex, 0, -1, -1, -1` in Fig. 3), resulting in iNumber not being incremented. Since numberOfInterfaces is a local (non-shared) variable the loop, the problem would persist in a concurrent setting, where the device may be disconnected while the loop is executed.

G Full Experimental Data

Table 3 presents our experimental results including timing information.

References

1. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time (preliminary report). In: STOC, ACM (1973) 1–9
2. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: FMICS. Number 66 in ENTCS, Elsevier (2002)
3. Jussila, T., Biere, A.: Compressing BMC encodings with QBF. ENTCS **174** (2007) 45–56
4. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI. Number 2937 in LNCS, Springer (2004) 239–251
5. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
6. Post, E.L.: The two-valued iterative systems of mathematical logic. Annals of Mathematics Studies (1941) 201–213

#	Manual	SAT		Seneschal		Rankfinder		QBF [-1,+1]		QBF $P(C,\mathcal{X})$		Biere et al. [2]	
1	L	64.77	●	M/O	–	0.09	○	T/O	–	T/O	–	T/O	–
2	L	706.98	●	41.45	●	13.26	●	T/O	–	T/O	–	T/O	–
3	L	2.76	●	11.96	●	0.08	○	9.65	●	T/O	–	T/O	–
4	L	9.32	●	9.39	●	1.00	●	150.56	●	T/O	–	28.46	●
5	N	0.53	○	14.12	○	0.06	○	0.82	○	T/O	–	T/O	–
6	N	4.19	○	M/O	–	0.06	○	2.13	○	T/O	–	T/O	–
7	N	19.53	○	M/O	–	0.08	○	T/O	–	T/O	–	T/O	–
8	L	9.29	●	8.12	●	1.05	●	567.73	●	T/O	–	T/O	–
9	T	0.28	○	7.17	○	0.12	◐	1129.88	○	T/O	–	T/O	–
10	N	19.95	○	9.52	○	0.07	○	2.89	○	T/O	–	0.06	○
11	T	19.53	○	21.06	○	0.08	○	T/O	–	T/O	–	T/O	–
12	L	2.03	●	7.83	●	0.84	●	6.97	●	T/O	–	93.12	●
13	L	6.70	●	9.14	●	0.08	○	T/O	–	T/O	–	213.23	●
14	N	0.80	○	7.65	○	0.12	◐	T/O	–	T/O	–	T/O	–
15	T	0.85	○	7.43	○	0.62	◐	T/O	–	T/O	–	M/O	–
16	L	193.72	●	M/O	–	0.68	◐	T/O	–	T/O	–	T/O	–
17	L	7.18	●	17.83	●	0.07	○	T/O	–	T/O	–	T/O	–
18	L	26.84	●	130.60	●	1.68	●	T/O	–	T/O	–	T/O	–
19	L	4.10	●	5.48	●	0.15	◐	T/O	–	T/O	–	T/O	–
20	L	0.96	●	6.39	●	0.07	○	10.51	●	T/O	–	3.21	●
21	T	23.63	○	0.21	○	0.20	○	T/O	–	T/O	–	T/O	–
22	L	0.98	●	T/O	–	0.07	○	24.27	●	T/O	–	T/O	–
23	L	13.76	●	23.97	●	12.99	●	T/O	–	T/O	–	7.25	●
24	L	4.85	●	11.96	●	1.98	●	T/O	–	T/O	–	T/O	–
25	L	1.07	●	5.84	●	0.12	◐	T/O	–	T/O	–	T/O	–
26	L	9.13	●	5.65	●	0.16	◐	T/O	–	T/O	–	T/O	–
27	L	162.40	●	M/O	–	672.55	●	T/O	–	T/O	–	T/O	–
28	N	0.26	○	6.47	○	0.05	○	0.24	○	T/O	–	26.64	○
29	T	0.79	○	7.21	○	T/O	–	T/O	–	T/O	–	T/O	–
30	L	1.88	●	T/O	–	0.08	○	T/O	–	T/O	–	T/O	–
31	T	1.39	○	T/O	–	0.07	○	0.66	○	T/O	–	9.37	●
32	T	7.62	○	0.27	○	0.23	○	T/O	–	T/O	–	T/O	–
33	L	1.03	●	7.54	●	0.13	◐	T/O	–	T/O	–	T/O	–
34	L	T/O	–	M/O	–	T/O	–	T/O	–	T/O	–	T/O	–
35	N	173.28	○	M/O	–	0.70	○	T/O	–	T/O	–	0.33	○
36	L	1.03	●	7.75	●	0.41	●	T/O	–	T/O	–	7.84	●
37	T	0.80	○	7.53	○	0.17	◐	T/O	–	T/O	–	T/O	–
38	L	2.19	●	8.73	●	0.96	●	T/O	–	T/O	–	T/O	–
39	L	9.06	●	T/O	–	5.59	●	T/O	–	T/O	–	M/O	–
40	L	0.37	●	97.15	●	0.07	○	0.61	●	T/O	–	48.29	●
41	L	0.99	●	8.12	●	0.07	○	14.56	●	T/O	–	T/O	–
42	L	7.02	●	23.22	●	4.10	●	T/O	–	T/O	–	T/O	–
43	L	11.95	●	12.15	●	9.58	●	11.06	●	T/O	–	T/O	–
44	N	18.96	○	M/O	–	0.11	○	2.87	○	T/O	–	0.10	○
45	T	0.85	○	9.50	○	0.07	○	T/O	–	T/O	–	2.67	●
46	L	0.89	●	7.40	●	0.76	●	T/O	–	T/O	–	0.61	●
47	L	0.33	●	5.76	●	0.11	◐	0.66	●	T/O	–	138.02	●
48	T	4.17	○	8.23	○	0.07	○	T/O	–	T/O	–	T/O	–
49	T	0.72	○	8.01	○	0.08	○	0.76	○	T/O	–	T/O	–
50	T	0.71	○	7.01	○	0.04	○	0.78	○	T/O	–	1941.30	●
51	L	40.49	●	61.27	●	42.67	●	T/O	–	T/O	–	93.40	●
52	T	0.77	○	6.63	○	0.14	◐	1.00	○	T/O	–	T/O	–
53	L	43.34	○	M/O	–	0.12	○	T/O	–	T/O	–	T/O	–
54	N	0.81	○	7.22	○	M/O	–	T/O	–	T/O	–	T/O	–
55	L	0.76	●	8.31	●	0.82	●	2617.80	●	T/O	–	T/O	–
56	L	53.39	●	6.22	●	2.44	●	T/O	–	T/O	–	T/O	–
57	L	7.90	●	23.71	●	8.76	●	T/O	–	T/O	–	M/O	–
58	L	84.43	●	T/O	–	0.63	◐	T/O	–	T/O	–	T/O	–
59	L	0.52	●	T/O	–	0.08	○	575.37	●	816.13	●	T/O	–
60	N	0.78	○	9.78	○	0.05	○	0.83	○	T/O	–	0.05	○
61	T	79.19	○	T/O	–	0.13	○	T/O	–	T/O	–	1.51	●

L Terminating, and linear ranking functions exist.	T Terminating (non-linear)
	N Non-terminating
● Termination was proven	◐ Incorrect under bit-vector semantics
○ (Possibly) Non-terminating	'–' Memory or time limits exhausted

Table 3. Experimental Data