FMSD manuscript No.
(will be inserted by the editor)

Ranking Function Synthesis for Bit-Vector Relations

Byron Cook · Daniel Kroening · Philipp Rümmer · Christoph M. Wintersteiger

Received: date / Accepted: date

Abstract Ranking function synthesis is a key component 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.

This is is an extended version of our TACAS 2010 paper [11]. 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, the EU FP7 STREP MOGENTES, the ARTEMIS CESAR project, and ERC project 280053.

Byron Cook
Microsoft Research, Cambridge, UK
E-mail: bycook@microsoft.com

Daniel Kroening
Oxford University, Oxford, UK
E-mail: kroening@cs.ox.ac.uk

Philipp Rümmer
Uppsala University, Uppsala, SE
E-mail: philipp.ruemmer@it.uu.se

Christoph M. Wintersteiger
Microsoft Research, Cambridge, UK
E-mail: cwinter@microsoft.com

1 Introduction

Many termination provers for imperative programs compose termination arguments by repeatedly invoking ranking function synthesis tools (instances are [7, 10, 14, 24]). Such synthesis tools are available for programs formulated with the help of various logical theories, including linear and non-linear arithmetic, arrays, or heap. 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, even if they are in a fragment that could be handled in a sound and complete fashion, such as finite-state programs over bit-vectors.

This article 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 method based on the reduction of bit-vectors to *Presburger arithmetic,* in combination with a novel and complete synthesis algorithm of linear ranking functions for transition relations defined in Presburger arithmetic; and
(ii) a *template-matching approach* for predefined classes of ranking functions, here instantiated for linear ranking functions. We make use of efficient QBF- and SAT-techniques in order to synthesise ranking functions from templates.

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 [24], which uses rational arithmetic. We also compare the performance of our methods with an approach to termination checking that is not based on ranking functions, the rewriting of termination properties to safety properties according to Biere et al. [6]. Our experimental results indicate that, on practical examples, the new methods presented in this article clearly surpass known methods in terms of precision and performance.

Contribution We introduce two new methods for ranking function synthesis for bit-vector programs: an extension of the approach in [24] to transition relations defined in Presburger arithmetic, and a template-matching method instantiated for linear ranking functions. Both methods are shown to be sound and complete for the computation of linear ranking functions of bit-vector relations. We give an extensive theoretical and empirical evaluation of our methods. Through an experimental comparison to pre-existing techniques, we demonstrate the practicality of our methods and identify reasons for impracticality of previous approaches.

Organisation of the Article In Sect. 2, we define syntax and semantics of the programs we consider, briefly explain the architecture of termination provers, and provide motivating examples. In Sect. 3, a known approach for ranking function synthesis based on linear programming is discussed. Subsequently, a new extension to this method is presented that handles bit-vector programs soundly. Sect. 4 describes how linear ranking functions for bit-vector programs can be defined in terms of affine geometry, which gives rise to a new approach based on template-matching for predefined classes of ranking functions, described in Sect. 5. In Sect. 6, the results of an experimental evaluation of all new methods are given and compared to results obtained through known approaches.

2 Bit-Vector Programs and Termination Analysis

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 (a preprocessing stage of) the termination prover (or model checker).

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*); this is not a restriction, as will become clear in the next section. Apart from this, our program syntax permits guards ($\mathsf{assume}\ (t)$), sequential composition ($\beta;\gamma$), choice ($\beta \mathrel{\square} \gamma$), 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 (unsigned) bit-vectors of width $\alpha(x) > 0$. The syntactic categories of programs, statements, and expressions are defined by the following grammar:

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

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 $*_n$ 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

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

Fig. 1 Code fragment of Windows driver audio/gfxswap.xp/filter.cpp (#14 in our evaluation)

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 usual. 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). The $<_s$ operator could in principle be expressed by means of reduction to the unsigned operator $<_u$, but keeping $<_s$ allows us to simplify presentation of later sections. Adding further operations, e.g., signed division or bit-vector concatenation, is straightforward.

Example 1 We consider the program given in Figure 1. The program contains an unsigned 32-bit variable `ulByteCount`, as well as a signed 32-bit variable `nLoop`. Recasting the program in unsigned arithmetic, with the help of a fresh (unsigned) variable x with $\alpha(x) = 32$, and $-4 \ \equiv\ 2^{32} - 4 \mod 2^{32}$, the bit-vector program for a single loop iteration is

$$\mathsf{assume}\ (\neg(x = 0_{32}));\ x := x + (2^{32} - 4)_{32}\ . \tag{1}$$

Typing Rules for Bit-Vector Programs In the whole article, we assume that considered bit-vector expressions and programs are well-typed, in particular that operands in expressions have correct bit-width. This requirement will also be important for the complexity theorem given in Sect. 2.2, where assumptions about the bit-widths of all expressions in a program have to be made. 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:

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

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

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

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

Formal Semantics of Bit-Vector Programs 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 set of possible values of a well-typed expression $t : n$, evaluated in state $s \in \mathcal{S}$, is denoted by $val_s(t)$ and defined recursively by equations like the following:

$$\begin{aligned}
val_s(*_n) &= \mathbb{B}^n \\
val_s(x) &= \{s(x)\} \subset \mathbb{B}^n \\
val_s(0_n) &= \{\langle 0, 0, \ldots, 0, 0\rangle\} \subset \mathbb{B}^n \\
val_s(1_n) &= \{\langle 0, 0, \ldots, 0, 1\rangle\} \subset \mathbb{B}^n \\
&\vdots \\
val_s(t_1 \circ t_2) &= \{a_1 \circ a_2 \mid a_1 \in val_s(t_1), a_2 \in val_s(t_2)\} \subseteq \mathbb{B}^n
\end{aligned}$$

In the last equation, it is assumed that a concrete definition of every binary operation $\circ$ on bit-vectors $a_1, a_2 \in \mathbb{B}^n$ is available.

The transition relation induced by a well-typed statement β is denoted by $R_\beta \subseteq \mathcal{S} \times \mathcal{S}$, and is again defined recursively:

$$\begin{aligned}
R_{\mathsf{skip}}(s, s') &\equiv s = s' \\
R_{\mathsf{assume}\ (t)}(s, s') &\equiv s = s' \wedge val_s(t) = \{1\} \\
R_{\beta\ \Box\ \gamma}(s, s') &\equiv R_\beta(s, s') \vee R_\gamma(s, s') \\
R_{\beta;\gamma}(s, s') &\equiv \exists s'' \in \mathcal{S}.\ R_\beta(s, s'') \wedge R_\gamma(s'', s') \\
R_{x:=t}(s, s') &\equiv s'(x) \in val_s(t) \wedge (\forall y \in \mathcal{X} \setminus \{x\}.\ s(y) = s'(y)) \\
R_{\beta\ \mathsf{repeat}\ \{\,\gamma\,\}}(s, s') &\equiv \exists s'' \in \mathcal{S}.\ R_\beta(s, s'') \wedge R^*_\gamma(s'', s')
\end{aligned}$$

In the definitions, and everywhere in the article, R^* denotes the reflexive transitive closure of a binary relation R, and $\equiv$ states logical equivalence of two formulae.

2.2 The Termination Problem and its Complexity

Bit-vector programs do not provide any heap or recursion and therefore belong to the class of *constant memory* programs, which means that the memory consumption is defined upfront and does not depend on program inputs. We say that a given bit-vector program β repeat $\{\,\gamma\,\}$ *terminates* if the transition relation $R_{\beta\ \mathsf{repeat}\ \{\,\gamma\,\}}$ is *well-founded,* in other words, if there is no infinite sequence of states $s_0, s_1, s_2, \ldots \in \mathcal{S}$ with $R_{\beta\ \mathsf{repeat}\ \{\,\gamma\,\}}(s_i, s_{i+1})$ for all $i \geq 0$.

Example 2 We assume $\alpha(n) = 8$. Of the following bit-vector programs, (2) terminates, while the other two programs are non-terminating:

$$\mathsf{assume}\ (n = 0_8);\ \mathsf{repeat}\ \{\, \mathsf{assume}\ (n <_u 250_8);\ n := n + 1_8 \,\} \tag{2}$$

$$\mathsf{assume}\ (n = 0_8);\ \mathsf{repeat}\ \{\, \mathsf{assume}\ (n <_u 255_8);\ n := n + 2_8 \,\} \tag{3}$$

$$\mathsf{skip};\ \mathsf{repeat}\ \{\, \mathsf{assume}\ (n <_u 10_8) \,\} \tag{4}$$

Termination of bit-vector programs is decidable, more precisely, the termination problem is PSPACE-complete: polynomial memory is needed in the size of the program. For this complexity result, it is necessary to bound the size of bit-vector expressions and variables occurring in a program (also see [22]). Given any positive integer B, we assume that $\mathcal{P}_B$ is the class of bit-vector programs in which the bit-width of all variables and expressions is at most B (as derived by the typing rules in the previous section).

Lemma 1 *For every $B \geq 1$, deciding termination of bit-vector programs in $\mathcal{P}_B$ is PSPACE-complete.*

Proof We first show PSPACE hardness and then membership in PSPACE.

Termination is PSPACE-hard We show that the termination problem for bit-vector programs is PSPACE-hard by a polynomial reduction of the satisfiability problem of QBF-formulae (which is the canonical PSPACE-complete problem [30]). 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 ψ is an expression of bit-width 1 in the grammar defined in Sect. 2.1, which is no restriction because the language provides the Boolean operators $\&$, $|$, $\neg$.

We need further variables to check satisfiability of ϕ: variables $r_1, \ldots, r_{n+1}$ with $\alpha(r_i) = 1$, where each r_i will be used to store the truth value of the sub-formula $Q_i x_i . \cdots Q_n x_n . \psi$; variables $\mathit{state}_1, \ldots, \mathit{state}_n$ with $\alpha(\mathit{state}_i) = 2$,[1] for the current assignment of each quantified variable; and finally, variables $\mathit{level}_0, \ldots, \mathit{level}_{n+1}$ with $\alpha(\mathit{level}_i) = 1$, to store which of the quantifiers is currently being processed.

The satisfiability checker has the following form (for sake of brevity, we omit the bit-widths of literals like 0_1):

```
$level_0 := 0;$ $level_1 := 1;$ $level_2 := 0;$ $\cdots$ ; $level_{n+1} := 0;$
$state_1 := 0;$ $\cdots$ ; $state_n := 0;$
repeat {    assume ($level_0$ & $\neg r_1$)
            $\Box$ $loop_1$ $\Box$ $loop_2$ $\Box$ $\cdots$ $\Box$ $loop_n$
            $\Box$ (assume ($level_{n+1}$); $r_{n+1} := \psi$; $level_{n+1} := 0$; $level_n := 1$) }
```

Note that the program will not terminate if it ever enters a state such that level_0 & $\neg r_1$, since the first assume statement does not have any side-effect. In this situation, level_0 records that the whole formula has been processed, and

[1] Alternatively, pairs $(\mathit{state}_i, \mathit{state}'_i)$ of variables with width $\alpha(\mathit{state}_i) = \alpha(\mathit{state}'_i) = 1$ can be used.

$\neg r_1$ that the formula evaluated to 0. Each of the blocks $loop_i$ is responsible for enumerating the possible values of x_i and evaluating the quantifier Q_i:

```
assume $(level_i)$; $level_i := 0$;
  (   (assume $(state_i = 0)$; $state_i := 1$; $level_{i+1} := 1$; $x_i := 0$)
    □ (assume $(state_i = 1)$; $state_i := 2$; $level_{i+1} := 1$; $r_i := r_{i+1}$; $x_i := 1$)
    □ (assume $(state_i = 2)$; $state_i := 0$; $level_{i-1} := 1$; $r_i := r_i \circ r_{i+1}$)   )
```

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

We can observe that the size of each $loop_i$ is constant (independent of n). The size of all $loop_i$ blocks together is therefore in $O(n)$, and the size of the whole satisfiability checker is in $O(s)$, where s is the size of the formula ϕ.

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 achieved in polynomial time.

Termination is in PSPACE To prove that the termination problem for bit-vector programs is in PSPACE, we encode the termination problem for a program α into a QBF-formula of polynomial size in the size of α. Because the satisfiability of QBF-formulae is in PSPACE [30], this shows that program termination is in PSPACE as well. The construction is based on the classical proof that QBF is PSPACE-complete [30] 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, and B). We then recursively define a predicate $reach(a, b, n)$ 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 $reach(a, b, n)$ is:

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

Expanding $reach(a, b, n)$ 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:

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

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

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, the size of β, γ, and B, 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 [6]), and subsequent application of the QBF-based Bounded Model Checking technique introduced in [20]. $\square$

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.

2.3 Ranking Functions and the Terminator Algorithm

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 article is the well-founded domain of natural numbers $(\mathbb{N}, <)$. In general, we can directly conclude:

Lemma 2 *If there exists a ranking function m for the transition relation R_β of a program β, then β terminates.*

Proof Suppose β does not terminate, which means that there is an infinite sequence of states $s_0, s_1, s_2, \ldots \in \mathcal{S}$ such that $R_\beta(s_i, s_{i+1})$ for all $i \geq 0$. By Def. 1, this implies that $m(s_{i+1}) \prec m(s_i)$ for all $i \geq 0$, contradicting the assumption that $(D, \prec)$ is a well-founded domain. $\square$

Program termination can therefore be shown with the help of ranking functions. By the disjunctive well-foundedness theorem [25], this is simplified to the problem of finding a ranking function for every cycle through a program β. The ranking functions $m_1, m_2, \ldots, m_n$ found for n cyclic paths are used to construct a global, disjunctive *ranking relation*

$$M(a, b) \ = \ \bigvee_{i=1}^{n} m_i(b) \prec m_i(a)\ . \tag{5}$$

Although M is in general not a well-founded relation, it can be shown that the existence of M nevertheless implies the termination of β [25].

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

Fig. 2 Code fragment of Windows driver kernel/agplib/init.c (#40 in our evaluation)

One technique that puts this theorem to use is the *Terminator* Algorithm [12, 13]. In this approach, termination of a program is first expressed as a *safety* property [6], initially assuming that no control state of the program is visited repeatedly. Consequently, a software model checker is applied to obtain a counterexample, i.e., an example of a recurring control state. This counterexample consists of a *stem* β leading to a *cycle* γ in the control flow graph of the program. What follows is an analysis solely concerned with the program β `repeat` $\{\, \gamma \,\}$ consisting of the stem and the cycle, which is why we may safely restrict ourselves to single-loop programs here. For further details, consult [6].

The next step in the procedure is to synthesise a ranking function for γ, which can be seen as a straight-line program, i.e., a program without loops or choices. If a ranking function m_γ can be found for the transition relation R_γ, the original safety property is weakened to only search for recurring control states with the property that $m_\gamma(s') \not< m_\gamma(s)$ for the full program states s, s', and the process starts over. This means that, incrementally, a disjunctive ranking relation (5) is constructed. If no further cycles are found, termination of the program is proven.

2.4 Arithmetic Intricacies in Termination Analysis

We discuss three examples extracted from Windows device drivers that illustrate the difficulty of termination checking for low-level code, in this case in ANSI-C. These examples will be revisited in later sections to illustrate our methods.

The first example (Figure 2) contains a while loop that iterates as long as bits are set in the variable `i` (this method to clear bits in an integer number goes back to [31]). 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 easy to achieve 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 cannot decrease indefinitely as it is bounded from below ($\mathtt{i} > 0$).

The second program (Figure 1) is non-terminating, because the variable `nLoop` might be initialised with a value that is not a multiple of four, 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

```
unsigned char Index;
unsigned int Head, i;

assume(Index != ((Head - 1) & 31));
i = Head;
while (i!=Index)
  i = (i+1) & 31;
```

Fig. 3 Code fragment of the driver audio/ac97/wavepcistream2.cpp (#5 in our evaluation)

four. 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 $m(\texttt{nLoop}, \texttt{ulByteCount}) = \texttt{nLoop}$.

Figure 3 contains another example of potentially non-terminating bit-vector code. This code does not terminate when `Index` > 31, because some of the upper bits of `Index` are set, but can never be set in `i`.

3 Synthesis of Ranking Functions by Linear Programming

3.1 Preliminaries

The ranking function synthesis method underlying termination provers like Terminator [13] or ARMC [26] was developed by Podelski and Rybalchenko [24]. In their setting, ranking functions are generated for transition relations of the form $R \subseteq \mathbb{Q}^n \times \mathbb{Q}^n$ which are described by a system of linear inequalities, i.e.,

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

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 fixed 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

$$\forall x, x' \in \mathbb{Q}^n\ .\ R(x, x') \text{ implies } rx + c \geq 0 \wedge rx' + c \geq 0 \wedge rx' + \delta \leq rx\ . \quad (6)$$

Coefficients r and c for which this implication is satisfied can be constructed with the help of conditions provided by Farkas' lemma, of which the 'affine' form given in [29] is appropriate:

Lemma 3 (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, $c \in \mathbb{Q}^k$ is a (row) vector, and $\delta \in \mathbb{Q}$ is a rational. Then*

$$\{x \in \mathbb{Q}^k : Ax \leq b\} \subseteq \{x \in \mathbb{Q}^k : cx \leq \delta\} \qquad (7)$$

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

Using this lemma, a necessary and sufficient criterion for the existence of linear ranking functions can be formulated as follows. (For details regarding the connection between the coefficients in the ranking functions and Farkas' lemma we refer the reader to the proof by Podelski and Rybalchenko [24, Theorem 2], as well as to the proof of Lem. 5 below.)

Theorem 1 (Existence of linear ranking functions [24]) *Suppose that $A, A' \in \mathbb{Q}^{n\times k}$ are matrices, $b \in \mathbb{Q}^n$ is a vector, and $R(x, x') \equiv Ax + A'x' \leq b$ is a non-empty transition relation. The relation R has a linear ranking function $m(x) = rx + c$ iff there are non-negative (row) vectors $\lambda_1, \lambda_2 \in \mathbb{Q}^n$ such that:*

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

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 exist relations $R(x, x') \equiv Ax + A'x' \leq b$, with $x, x' \in \mathbb{Z}^k$ ranging over integers, for which linear ranking functions exist, but the criterion in Theorem 1 fails. An example for this situation is:

$$R(x, x') \;\equiv\; 0 \leq x \leq 4 \;\wedge\; 9 \leq 10x' - 2x \leq 11 \;.$$

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. A non-terminating sequence $x_0, x_1, x_2, \ldots$ of program states is, for instance, defined by the recurrence equations $x_0 = 0$ and $x_{i+1} = x_i + 0.2^i$. Since $R(1.25, 1.25)$, an even simpler counterexample to termination is the sequence 0.0, 1.1, 1.25, 1.25, 1.25, ...

3.2 Bit-Vector Ranking Functions through 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.

3.2.1 Linear Ranking Functions over the Integers

The previous section considered transition relations expressed as conjunctions of inequalities, which can only describe *convex* relations (i.e., $R(x, x')$ and $R(y, y')$ together imply $R(\lambda x + (1-\lambda)y,\; \lambda x' + (1-\lambda)y')$ for every $\lambda \in [0, 1]$). Convexity is often violated by bit-vector operations, for instance by operations that exhibit overflow/wrap-around behaviour. For instance, over the domain $\{0, 1, 2, 3\}$ (calculating modulo 2^2) we have $1_2 + 1_2 = 2_2$ and $3_2 + 3_2 = 2_2$, but $2_2 + 2_2 = 0_2$. Non-convex operations can naturally be encoded with the help of disjunctive constraints, which means that we have to consider transition relations of the form

$$R(x, x') \;\equiv\; \bigvee_{i=1}^{l} A_i x + A'_i x' \leq b_i \;, \tag{8}$$

where $l \in \mathbb{N}$, $A_i, A'_i \in \mathbb{Z}^{n \times k}$, $b_i \in \mathbb{Z}^n$, and $x, x' \in \mathbb{Z}^k$ range over integer vectors. Linear ranking functions for such relations can be constructed by solving an implication like (6) for each disjunct of the relation, as shown below.

Both Podelski and Rybalchenko's method [24] and the method described in this section rely on Farkas' lemma. In the world of integers and bit-vectors, however, only one of the implications stated in the lemma holds: if x in (7) ranges over the integers, implied inequalities 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 $\{x \in \mathbb{Q}^k \mid Ax \leq b\}$ coincides with its integral hull (the convex hull of the integer points contained in it).[2] For our purposes, we therefore need the following, slightly modified version of Farkas' lemma:

Lemma 4 (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*

$$\{x \in \mathbb{Z}^k : Ax \leq b\} \subseteq \{x \in \mathbb{Z}^k : cx \leq \delta\} \tag{9}$$

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

The difference between Lem. 4 and the rational Farkas' lemma (Lem. 3) is the assumption that $Ax \leq b$ is integral, and the use of $\mathbb{Z}$ instead of $\mathbb{Q}$ in (9).

Proof We show that (7) if and only if (9) in the case of an integral system $Ax \leq b$. The conjecture then follows by Lem. 3.

(7) $\Rightarrow$ (9): holds because of $\mathbb{Z} \subset \mathbb{Q}$.

(9) $\Rightarrow$ (7): suppose (9) 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. This implies (7). □

[2] This deviates from the terminology in [29], 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.

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 [29]. 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

$$\max \{cx : A'x \leq b'\} = \min \{yb : y \geq 0, \, yA' = c\}$$

has an integral optimum solution y for each integral vector c for which the minimum is finite [29]. $A'x \leq b'$ can then be strengthened to $A'x \leq \lfloor b' \rfloor$ without losing integer solutions. The resulting system $A'x \leq \lfloor b' \rfloor$ can again be transformed to a total dual integral system, and strengthened, etc. By iterating this refinement loop, in at most exponentially many steps an integral system of inequalities is derived (this follows from Theorem 17.4 in [29]).

We are now in the position to give a criterion for the existence of ranking functions for disjunctive linear systems over integers:

Lemma 5 *Suppose $l \in \mathbb{N}$, and suppose that for each $i \in \{1, \ldots, l\}$ a pair of matrices $A_i, A'_i \in \mathbb{Q}^{n_i \times k}$ and a vector $b_i \in \mathbb{Q}^{n_i}$ are given such that the system $A_i x + A'_i x' \leq b_i$ is satisfiable and integral. The disjunctive transition relation*

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

has a linear ranking function $m(x) = rx + c$ if and only if there are non-negative (row) vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^n$ for $i \in \{1, \ldots, l\}$ such that

$$\lambda_1^i A'_i = 0, \quad \lambda_2^i (A_i + A'_i) = 0, \quad \lambda_2^i b_i < 0, \quad (\lambda_1^i - \lambda_2^i) A_i = 0, \quad \lambda_2^i A'_i = r \; . \tag{10}$$

Proof $\Rightarrow$: Assume the relation $R(x, x')$ has a ranking function $m(x) = rx + c$. Arguing as in the proof [24, Theorem 2], this means that for some $\delta > 0$ and all $i \in \{1, \ldots, l\}$ we have:

$$\text{for all } x, x' \in \mathbb{Z}^k : A_i x + A'_i x' \leq b_i \text{ implies} \qquad\qquad$$
$$rx + c \geq 0 \land rx' + c \geq 0 \land rx' + \delta \leq rx \tag{11}$$

By Lem. 4, this implies that there are non-negative vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^n$ such that for $i \in \{1, \ldots, l\}$:

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

It is now easy to see that (10) is implied by these equations and inequalities.

$\Leftarrow$: Assume (10) holds for non-negative vectors $\lambda_1^i, \lambda_2^i \in \mathbb{Q}^n$ 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 $m_i(x) = r_i x + c_i$. Due to the last equation in (10), we have $r_i = r$ for all $i \in \{1, \dots, l\}$, which implies that

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

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

3.2.2 Ranking Functions for Presburger Arithmetic

Presburger arithmetic (PA) is the first-order theory of integer arithmetic without multiplication [27]. We describe a complete procedure to generate linear ranking functions for PA-defined transition relations by reduction to Lem. 5. Assuming that a polynomial method is used to solve (10), and that a transition relation is defined by a *quantifier-free* Presburger formula, the complexity of our procedure is singly exponential.

Suppose a transition relation $R(x, x')$ is defined by a Presburger formula. Because PA admits quantifier elimination [27], 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. 5, 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 normal form, replacing equations $s = t$ with inequalities $s \le t \wedge t \le s$, the relation $R'(x, y, x', y')$ can be stated as in (8):

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

We can then apply Lem. 5 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 can be added to (10) that constrain the corresponding entries of the vector r to zero.

Replacing Divisibility Constraints by Disjunctions of Equations The transformation from $R(x,x')$ to $R'(x,y,x',y')$ uses the following equivalences:

$$\begin{aligned}
&\epsilon \mid (cx + dx' + e) \\
&\equiv\ \epsilon \ \Big|\ \Big(cx - \epsilon\Big\lfloor\frac{cx}{\epsilon}\Big\rfloor + dx' - \epsilon\Big\lfloor\frac{dx'}{\epsilon}\Big\rfloor + e\Big) && (12)\\
&\equiv \bigvee_{\substack{i\in\mathbb{Z}\\ 0\le i\cdot\epsilon - e < 2\epsilon}} i\cdot\epsilon - e \ = \ cx - \epsilon\Big\lfloor\frac{cx}{\epsilon}\Big\rfloor + dx' - \epsilon\Big\lfloor\frac{dx'}{\epsilon}\Big\rfloor && (13)\\
&\equiv\ \exists y_c, y_d'.\ \begin{pmatrix} 0 \le cx - \epsilon y_c < \epsilon \wedge 0 \le dx' - \epsilon y_d' < \epsilon \\ \wedge\ (\bigvee_{0\le i\cdot\epsilon-e<2\epsilon} i\cdot\epsilon - e = cx - \epsilon y_c + dx' - \epsilon y_d') \end{pmatrix} && (14)
\end{aligned}$$

Equivalence (12) holds because divisibility is not affected by subtracting multiples of ϵ on the right-hand side, while (13) 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 (12) 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 (14).

To avoid the introduction of quantifiers, the quantified variables y_c, y_d' can be 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 $R(x,x')$:

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

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 $\bigvee_{0\le i\cdot\epsilon-e<2\epsilon} i\cdot\epsilon - e = cx - \epsilon y_c + dx' - \epsilon y_d'$ as in (14); this is possible regardless of whether $\epsilon \mid (cx + dx' + e)$ occurs in a positive or negative position (i.e., underneath negations). Iterating this procedure eventually leads to a transition relation $R'(x,y,x',y')$ without divisibility constraints, such that $\exists y, y'.R'(x,y,x',y') \equiv R(x,x')$.

Lemma 6 *The procedure described above decides the existence of linear ranking functions for transition relations $R(x,x')$ defined in quantifier-free PA in deterministic (singly) exponential time.*

Proof When eliminating equations and divisibility constraints in R by means of auxiliary variables, we first derive a transition relation $R'(x,y,x',y')$ that is at most polynomially bigger than $R(x,x')$. Rewriting to disjunctive normal form yields at most exponentially many disjuncts, each of which has size polynomial in the size of $R(x,x')$:

$$\begin{aligned}
R'(x,y,x',y') \ &\equiv\ \bigvee_{i=1}^{l} A_i\begin{pmatrix}x\\y\end{pmatrix} + A_i'\begin{pmatrix}x'\\y'\end{pmatrix} \le b_i \\
\exists y, y'.R'(x,y,x',y') \ &\equiv\ R(x,x') && (15)
\end{aligned}$$

For each of the disjuncts $A_i\binom{x}{y} + A'_i\binom{x'}{y'} \leq b_i$, satisfiability can be decided in exponential time, in order to remove unsatisfiable disjuncts. Furthermore, each of the disjuncts $A_i\binom{x}{y} + A'_i\binom{x'}{y'} \leq b_i$ can be transformed to an integral system (of at most exponential size) in exponential time (this follows from Theorem 17.4 in [29]).

Because of (15), 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. 5 to derive a ranking function $m(x, y) = rx + r'y + c$ of the relation $R'(x, y, x', y')$. By imposing the additional constraint $r' = 0$ (along with (10)), it can be ensured that the ranking function $m(x, y) = rx + c = m(x)$ only depends on x and not on y.

The system (10) of equations and inequalities has size exponential in the size of $R(x, x')$, and can be solved (over the rationals) in time singly exponential in the size of $R(x, x')$. □

3.2.3 Representation of Bit-vector Operations in Presburger Arithmetic

PA is expressive enough to capture the semantics of all bit-vector operations defined in Sect. 2, an observation that is frequently exploited in verification algorithms [8, 18, 23]. Thus, ranking functions for bit-vector programs can be generated using Lem. 6. For this, the domain $\mathbb{B}^n$ of bit-vectors of length n is identified with the subset $\{0, \ldots, 2^n - 1\}$ of integers, and bit-vector expressions can recursively be translated into equivalent Presburger formulae.

Suppose that r is a integer variable ranging over $\{0, \ldots, 2^n - 1\}$ (and therefore, equivalently, over the domain $\mathbb{B}^n$), and e is a bit-vector expression according to the grammar in Sect. 2.1. We define a binary function t in such a way that $t(r, e)$ is a PA formula that is equivalent to the bit-vector equation $r = e$, in particular:

$$\begin{aligned}
t(r, k_n) &= \exists \lambda. && \big(r = k + \lambda 2^n \wedge 0 \leq r < 2^n\big) \\
t(r, e + e') &= \exists r_e, r_{e'}. && \begin{pmatrix} t(r_e, e) \wedge t(r_{e'}, e') \wedge 0 \leq r < 2^n \wedge \\ (r = r_e + r_{e'} \vee r = r_e + r_{e'} - 2^n) \end{pmatrix} \\
t(r, \neg e) &= \exists r_e. && \big(t(r_e, e) \wedge r = 2^n - r_e\big)
\end{aligned}$$

The quantifiers used in the translation can subsequently be eliminated using standard procedures [27], resulting in a formula in quantifier-free PA. The translation of non-linear operations like $\times$ and $\&$ can be done in a similar manner by case analysis over the values of their operands. Such an encoding is possible because the variables of bit-vector programs range over finite domains of fixed size, albeit at the cost of a generally exponential blow-up in formula size. Nevertheless, we observed that the translation is well-behaved in many practical cases, e.g., when at least one operand of a non-linear operation ranges over a small interval of values (also see Sect. 6).

For the full details of the translation, we refer the interested reader to the (both human- and machine-readable) axioms used in our implementation

Seneschal,[3] which closely correspond to the cases of the definition of $t(r, e)$ shown here.

Example 3 We illustrate how the bit-vector program (1) from Example 1 (corresponding to Figure 1) can be translated to PA:

$$x \neq 0 \ \wedge \ \big(2^{32} \mid (x' - x - 2^{32} + 4)\big) \ \wedge \ 0 \leq x < 2^{32} \ \wedge \ 0 \leq x' < 2^{32}$$

From the side conditions, we read off that the term $x' - x - 2^{32} + 4$ has the range $[5 - 2^{33}, 3]$, so that the divisibility constraint can directly be split into two equations (auxiliary variables as in (14) are unnecessary in this particular example). With further simplifications, we can express the transition relation as:

$$\big(x' = x - 4 \ \wedge \ 0 \leq x' \ \wedge \ x < 2^{32}\big) \vee \big(x' = x + 2^{32} - 4 \wedge 0 < x \wedge x' < 2^{32}\big)$$

It is now easy to see that each disjunct is satisfiable and integral, which means that Lem. 5 is applicable. Because the conditions (10) are not simultaneously satisfiable for all disjuncts, no linear ranking function exists for the program.

The implementation Seneschal used in our experiments is able to carry out the translation from bit-vector expressions to (quantifier-free) PA fully automatically.

4 The Vector Space of Linear Ranking Functions

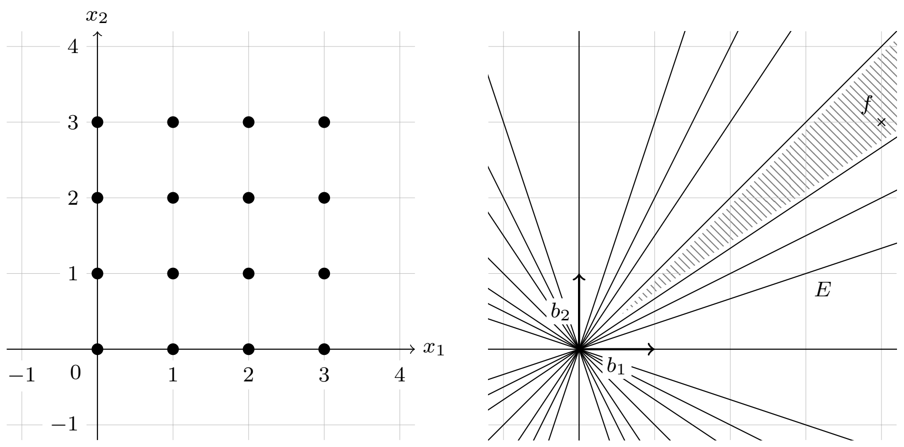
This section describes how linear ranking functions for bit-vector programs can be defined in terms of affine geometry. The characterization enables the derivation of bounds on the magnitude of coefficients in ranking functions, which we will exploit in a new, template-based ranking function synthesis method in Sect. 5.

4.1 Preliminaries

In the following, we consider an arbitrary transition relation $R(s, s')$ over a vocabulary $\mathcal{X}$ of variables. For sake of simplicity, it is assumed that the bit-width of all $|\mathcal{X}| = m$ variables is n, i.e., $\alpha(x) = n$ for all $x \in \mathcal{X}$. Given an arbitrary but fixed enumeration $x_1, \ldots, x_m$ of the variables $\mathcal{X}$, the states $s \in \mathcal{S}$ can be seen as the elements of the grid $\{0, 1, \ldots, 2^n - 1\}^m$, embedded in the vector space $\mathbb{Q}^m$, as illustrated in Figure 4(a).

Given this view on program states, it is clear that the candidates for linear ranking functions (for the transition relation R) are the elements of the set $V = \mathbb{Q}^m \to \mathbb{Q}$ of rational linear functions over the program variables $\mathcal{X}$. Indeed, every function $f \in V$ induces a strict partial order $\prec_f$ on the state

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



(a) State space $\mathcal{S}$ for variables x_1, x_2 with bit-width $n = 2$, embedded into $\mathbb{Q}^2$.

(b) The space V of linear ranking functions for the state space in (a).

Fig. 4 State space and ranking space for two 2-bit variables, including the set E of non-$\mathcal{S}$-injective functions. The grey area illustrates the class of functions that are order-equivalent to the $\mathcal{S}$-injective ranking function $f = 4b_1 + 3b_2$, i.e., $f(x_1, x_2) = 4x_1 + 3x_2$.

space $\mathcal{S}$, defined by $s \prec_f s' \equiv f(s) < f(s')$. Since $\mathcal{S}$ is finite, every such order is well-founded, and can be used as a termination argument for R if the implication $R(s, s') \Rightarrow s' \prec_f s$ holds for all s, s'.

We say that a function $f \in V$ is *$\mathcal{S}$-injective* if $f(s) \neq f(s')$ for all distinct states $s \neq s' \in \mathcal{S}$. Precisely the $\mathcal{S}$-injective functions give rise to (strict) *total* orders $\prec_f$ on the state space, while the orders induced by non-$\mathcal{S}$-injective functions are partial. We further say that two functions $f, f' \in V$ are *order-equivalent* if they induce the same order, i.e., $\prec_f = \prec_{f'}$. Obviously, as $\mathcal{S}$ is finite, V is partitioned into finitely many equivalence classes in this way.

Because functions f, f' in the same equivalence class can prove the termination of exactly the same transition relations, we can restrict the search for ranking functions to specific representatives from each equivalence class. This will be important when defining the template-based synthesis method in Sect. 5. Furthermore, it is enough to consider $\mathcal{S}$-injective functions: if the termination of a transition relation can be shown using a non-$\mathcal{S}$-injective ranking function, it can also be proven using an $\mathcal{S}$-injective ranking function.

4.2 The Geometry of Equivalence Classes

There is a simple geometric interpretation of the equivalence classes. The set V has the structure of a rational vector space, and is in fact isomorphic to the vector space $\mathbb{Q}^m$ into which the state space $\mathcal{S}$ is embedded (it is the *dual space* of $\mathbb{Q}^m$).

In the following paragraphs, it will be convenient to work with a standard basis of $\mathbb{Q}^m$ and V. Recall that states $s \in \mathcal{S}$ are mappings $\mathcal{X} \to \mathbb{B}^+$ from variables to bit-vectors, and for the purpose of this section can be considered as vectors $s = \big(s(x_1), s(x_2), \ldots, s(x_m)\big) \in \mathbb{Q}^m$ of rational numbers. We can then define the standard basis $\{s_1, \ldots, s_m\} \subseteq \mathcal{S}$, and its dual basis $\{b_1, \ldots, b_m\} \subseteq V$ with the help of the following equations (see, e.g., [29]):

$$s_i(x_j) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}, \qquad b_i(s_j) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}.$$

A non-$\mathcal{S}$-injective function f has the property that $f(s) = f(s')$ for some states $s \neq s'$. Because states are interpreted as vectors, and f is linear, this is equivalent to $f(s - s') = 0$. For any two states $s \neq s' \in \mathcal{S}$, 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-$\mathcal{S}$-injective functions is the finite union

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

of hyperplanes. The complement $P = V \setminus E$ is a finite union of open convex sets, each of which forms the interior of a convex (but unbounded) polyhedron. This is illustrated in Figure 4(b).

The interiors of these polyhedra coincide with the equivalence classes of $\mathcal{S}$-injective 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 $\mathcal{S}$-injective functions that are not order-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 $s' \prec_{f'} s$.

4.3 Representatives for Equivalence Classes

Functions $f \in V$ can (uniquely) be represented 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 rest of the section, we consider linear combinations with integer coefficients $\alpha_1, \ldots, \alpha_m$, and derive bounds on the absolute values of the coefficients such that elements from each equivalence class of $\mathcal{S}$-injective functions can be represented. These bounds will determine the number of bits needed in ranking function templates.

Theorem 2 *We fix a non-empty class $A \subseteq V$ of $\mathcal{S}$-injective 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.*

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

Proof To see that (i) holds, consider a plane $E_{s,s'} = \{f \in V \mid f(s - s') = 0\}$. We can choose the representation $f = \gamma_1 b_1 + \cdots + \gamma_m b_m$, and then expand the linear equation defining the hyperplane to

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

Since we assume that all variables have the bit-width n, the integer coefficients $v_i = s(x_i) - s'(x_i)$ in this 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 need to solve a system of $m-1$ such linear equations:

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

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 $\mathrm{sgn}(\sigma)$ is $+1$ for even and -1 for odd permutations σ):

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

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. □

In the next section, Theorem 2 will be used to establish the completeness of a template-based ranking function synthesis method.

5 Synthesis of Ranking Functions from Templates

A subset of all ranking functions for bit-vector programs can be identified by templates of a desired class of functions with undetermined coefficients. In order to find those coefficients, we consider two methods: (i) an encoding into quantified Boolean formulae (QBF) to check all suitable values, and (ii) a propositional SAT encoding 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. Using the ordering relation $<_s$ to interpret the output of $p(c, x)$ as a signed value, we formulate the following theorem, which provides a bound on w that guarantees that ranking functions can be represented for all programs that have linear ranking functions.

Theorem 3 *There exists a linear ranking function on path π with transition relation $R_\pi(x, x')$ if*

$$\exists c \,.\, \forall x, x' \,.\, R_\pi(x, x') \Rightarrow p(c, x') <_s p(c, x) \,. \tag{16}$$

Vice versa, if there exists a linear ranking function for π, then (16) *must be valid whenever*[4]

$$w \;\geq\; \max_i(\alpha(x_i)) \cdot (|\mathcal{X}| - 1) + |\mathcal{X}| \cdot \log_2 |\mathcal{X}| + 2 \,. \tag{17}$$

Proof The first half of the theorem is obvious. The second half of the theorem is shown using the observations made in Sect. 4. From part (ii) of Theorem 2, we can derive the number of bits needed for Theorem 3:

$$\begin{aligned} \lceil \log_2(2^{n(m-1)} \cdot m! + 1) \rceil \;&\leq\; n(m-1) + \log_2 m! + 1 \\ &\leq\; n(m-1) + m \log_2 m + 1 \end{aligned}$$

Because both positive and negative coefficients have to be represented, we need a further bit for the sign, which yields the bound $n(m-1) + m \log_2 m + 2$ given in Theorem 3. □

We illustrate that the number of bits required in the coefficients of ranking functions can indeed approach the bound given as right-hand side of (17).

[4] In [11], it was incorrectly stated that the constant term in (17) is "1" instead of "2".

Consider terminating programs (for which linear ranking functions exist) of the form

```
$x_1 := 1;$
repeat {
    assume $(x_1 \neq 0 \vee x_2 \neq 0 \vee x_3 \neq 0 \vee \cdots \vee x_{|\mathcal{X}|} \neq 0);$
    $x_{|\mathcal{X}|} := x_{|\mathcal{X}|} + (x_1 \div 255) \times (x_2 \div 255) \times \ldots \times (x_{|\mathcal{X}|-1} \div 255)$
    $\cdots$
    $x_3 := x_3 + (x_1 \div 255) \times (x_2 \div 255)$
    $x_2 := x_2 + (x_1 \div 255)$
    $x_1 := x_1 + 1$
}
```

where the bit-width of all variables and constants is n. Programs built after this scheme require ranking functions that order the program states in a lexicographic fashion. A suitable ranking function is of the form

$$p(c, x) = \cdots + c_3 \times x_3 + c_2 \times x_2 + c_1 \times x_1 \ ,$$

where $c_1 = -1$, $c_2 = -2^n$, $c_3 = -2^{2n}$, etc. This means that the corresponding bit-widths of the coefficients are $\alpha(c_1) = 2$, $\alpha(c_2) = n+2$, $\alpha(c_3) = 2n+2$, and in particular

$$\begin{aligned} \alpha(c_{|\mathcal{X}|}) &= (|\mathcal{X}| - 1) \cdot n + 2 \\ &\leq \max_i(\alpha(x_i)) \cdot (|\mathcal{X}| - 1) + |\mathcal{X}| \cdot \log_2 |\mathcal{X}| + 2 \ . \end{aligned}$$

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.

It is straightforward to flatten (16) 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 [5], sKizzo [4], and Squolem [21]. In our experiments, we use an experimental version of QuBE [17].

Despite much progress, the capacity of QBF solvers has not yet reached the level of efficacy 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 $\gamma_{c_i} \in \{0, 1, 3\}$ to each coefficient (corresponding to $\{-1, 0, 1\}$ in 2's complement). Let $\gamma_c = (\gamma_{c_1}, \ldots, \gamma_{c_{|\mathcal{X}|}})$. Negating and applying γ_c transforms (16) into

$$\neg\exists x, x' \ . \ R_\pi(x, x') \wedge \neg(p(\gamma_c, x') <_s p(\gamma_c, x)) \ , \qquad (18)$$

which is a bit-vector (or SMT *QF_BV*) 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_c until we find one for which (18) 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. 6.

Example 4 We consider the program given in Figure 2. The only variable in the program is `i`, and it is eight bits wide. We construct the polynomial

$$p(c, \mathtt{i}) = c \times \mathsf{cast}_9(\mathtt{i})$$

with $\alpha(c) = 9$. For the only path through the loop in this example, the transition relation $R_\pi(\mathtt{i,i'})$ is $\mathtt{i} \neq 0 \wedge \mathtt{i'} = \mathtt{i} \,\&\, (\mathtt{i} - 1)$. Solving the resulting formula

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

with a QBF-Solver does not return a result within an hour. We thus rewrite the formula according to (18) and obtain

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

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 $\mathsf{cast}_9(\mathtt{i})$ is a suitable ranking function. In this particular example, it is possible to omit the cast.

6 Experiments

6.1 Large-scale Benchmarks

Following Cook et al. [13], we implemented the Terminator algorithm to evaluate our ranking synthesis methods. Our implementation uses the SATABS model checker [9] as the reachability checker, which implements SAT-based predicate abstraction. Our benchmarks are device drivers from the Windows Driver Development Kit (WDK).[5] The WDK already includes verification harnesses for the drivers. We use GOTO-CC[6] to extract model files from a total of 87 drivers in the WDK. A subset of the results is presented in Table 1.

Our tool does not currently implement any techniques for arithmetic abstraction and consequently it is not able to find termination proofs for loops over singly and doubly-linked lists, which many drivers contain. Such abstractions can be automated by existing shape analysis methods (e.g., as presented by Yang et al. [33]).

[5] Version 6, now superseded by the Windows Driver Kit; see `http://msdn.microsoft.com/en-us/windows/hardware/gg581061`

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

hidusbfx2	cdrom	isousb	toaster-fitler	tape	sfilter	kbdclass	disk	bulkusb	Driver	
2	8	34	1	6	10	13	14	13	# total	Loops
2	4	0	1	3	10	7	7	13	# terminating	
0	4	11	0	3	0	5	7	0	# non-terminating	
2	0	7	1	1	0	2	2	6	# Rank Functions	
1	340	—	61	10	37	—	218	—	Time [min]	

Table 1 A selection of the results on full driver code. Notes: If ranking functions are successfully synthesised, but the final safety property is not proven within the time limit, the loop is classified as non-terminating. The entry '—' indicates that SATABS ran out of time (6 hrs) or memory (2GB), consequently the numbers of terminating and non-terminating loops do not add up to the total.

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 2 The behaviour of our implementation when run on the kbdclass driver.

Slicing the Input Just like Cook et al. [13], 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 termination prover analyses each loop separately and generates an inter-procedural slice [19] 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.[7] 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 keyboard class driver in the WDK (kbdclass) contains 13 loops in a harness (SDV_FLAT_HARNESS) that calls all dispatch functions nondeterministically.

Table 2 provides details on 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. 5 ('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

[7] 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).

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

Fig. 5 Code fragment from usb/bulkusb/sys/bulkpnp.c (simplified)

that no further counterexamples to termination exist. The entire analysis of this driver requires two hours. All experiments were run on 8-core Intel Xeon 3 GHz machines with 16 GB of RAM.

6.2 A Practical Termination Problem

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.

An excerpt of the driver code is shown in Figure 5. For every device, the driver requests an interface description structure to be searched. 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 documentation, it may return `NULL` if no interface matches the search criteria (`iIndex, 0, -1, -1, -1` in Figure 5, resulting in `iNumber` not being incremented. Since `numberOfInterfaces` is a local (non-shared) variable of the loop, the problem would persist in a concurrent setting, where the device may be disconnected while the loop is executed.

6.3 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). 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 sec, 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.[8] It is based on the prover Princess [28] for PA 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. 5 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.

The results of our evaluation are summarized in Table 3. The second 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. 5; Seneschal is the integer linear programming approach from Sect. 3.2.1; Rankfinder is the linear programming approach over rationals from Sect. 3.1; QBF [-1,+1] is a QBF template approach from Sect. 5 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, no linear ranking functions are found. On the original, unsliced programs, the SAT-based approach and Seneschal find suitable ranking functions, on benchmark #34 however, the model checker times out while attempting to prove that the function that was found is sufficient.

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 (the latter of which are reported as *possibly* non-terminating due to the incompleteness of the approach). It does not time out on any instance. Seneschal exhibits similar performance: it proves 31 programs as terminating, almost as many as the SAT-based template approach. It reports 25 benchmarks as (possibly) non-terminating and times out on five.

For the experiments using Rankfinder,[9] 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-

[8] `http://www.philipp.ruemmer.org/seneschal.shtml`
[9] `http://www7.in.tum.de/~rybal/rankfinder/`

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, with the consequence that a counterexample is not correctly excluded and subsequent abortion of the prover.

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

We also implemented the approach suggested by Biere et al. [6] (rightmost column of Table 3). This approach does not require synthesis of 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 four benchmarks (#31, #45, #50, and #61) that none of the other methods can handle as they require non-linear ranking functions.

In conclusion, our evaluation shows that the methods presented in this article outperform known approaches both in terms of runtime and precision. While existing approaches are sometimes able to synthesise non-linear ranking functions (e.g., Biere et al.), or they are sometimes able to find linear ranking functions faster (e.g., Rankfinder), their overall performance is greatly exceeded by our SAT- and LP-based approaches.

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

7 Related Work

Numerous efficient methods are now available for the purpose of finding ranking functions (e.g., [2,7,15,24]). Some tools are complete for the class of ranking functions for which they are designed (e.g., [24]), others employ a set of heuristics (e.g., [2]). Until now, no known tool has supported machine-level integers.

Wintersteiger et al. [32] provide a decision procedure for *quantified* bit-vector logic with uninterpreted functions (SMT $\mathcal{UFBV}$). This logic allows for a direct encoding of ranking function checks as

$$\exists f\ \forall x, x' \ .\ R_\pi(x, x') \Rightarrow f(x') < f(x)\ ,$$

where the range of f is a bit-vector of some pre-defined size. It has been demonstrated that when restricted to the same polynomial templates as presented in Section 5, their approach performs similar to our SAT-based enumeration approach while maintaining (relative) completeness. Their decision procedure

#	Manual	SAT		Seneschal		Rankfinder		QBF [-1,+1]		QBF $P(C,\mathcal{X})$		Biere et al. [6]	
1	L	52.07	●	17.67	●	0.05	○	–	–	–	–	–	–
2	L	1.16	●	22.02	●	1.19	●	–	–	–	–	–	–
3	L	0.30	●	10.97	●	0.03	○	49.45	●	567.79	●	–	–
4	L	0.29	●	7.60	●	0.37	●	16.28	●	220.65	●	10.95	●
5	N	0.18	○	15.15	○	0.04	○	0.78	○	–	–	–	–
6	N	0.18	○	20.33	○	0.03	○	1.57	○	–	–	–	–
7	N	17.93	○	–	–	0.03	○	–	–	–	–	–	–
8	L	0.40	●	8.36	●	0.36	●	–	–	–	–	–	–
9	T	0.12	○	8.05	○	0.08	◐	–	–	–	–	–	–
10	N	0.25	○	9.62	○	0.02	○	0.78	○	–	–	0.04	○
11	T	0.28	○	11.54	○	0.04	○	–	–	–	–	–	–
12	L	0.25	●	7.57	●	0.31	●	2.45	●	18.75	●	50.32	●
13	L	0.28	●	8.68	●	0.04	○	–	–	–	–	88.27	●
14	N	0.28	○	7.88	○	0.04	◐	–	–	–	–	–	–
15	T	0.57	○	14.82	○	0.27	◐	–	–	–	–	–	–
16	L	1.65	●	12.12	●	0.51	●	–	–	–	–	–	–
17	L	1.10	●	16.86	●	0.26	○	–	–	–	–	–	–
18	L	9.88	●	14.54	●	0.68	●	–	–	–	–	–	–
19	L	0.38	●	7.47	●	0.16	●	–	–	–	–	–	–
20	L	0.31	●	8.56	●	0.01	○	–	–	–	–	1.09	●
21	T	8.09	○	0.07	○	0.06	○	–	–	–	–	–	–
22	L	0.36	●	–	–	0.02	○	–	–	–	–	–	–
23	L	0.44	●	14.09	●	0.48	●	13.81	●	570.24	●	1.09	●
24	L	0.60	●	8.36	●	0.69	●	–	–	–	–	–	–
25	L	0.35	●	7.64	●	0.18	●	–	–	–	–	–	–
26	L	0.38	●	7.70	●	0.20	●	–	–	–	–	–	–
27	L	1.65	○	16.36	○	0.20	◐	–	–	–	–	–	–
28	N	0.08	○	8.95	○	0.03	○	0.24	○	–	–	9.02	○
29	T	0.29	○	8.15	○	–	–	–	–	–	–	–	–
30	L	0.30	●	–	–	0.02	○	1735.81	●	–	–	–	–
31	T	0.10	○	23.16	○	0.03	○	0.25	○	–	–	2.46	●
32	T	1.00	○	6.04	○	0.10	○	0.79	○	–	–	–	–
33	L	0.39	●	7.52	●	0.16	●	–	–	–	–	–	–
34	L	1114.95	○	217.93	○	0.05	○	–	–	–	–	–	–
35	N	0.36	○	16.07	○	0.39	○	–	–	–	–	–	–
36	L	0.32	●	7.43	●	0.20	●	–	–	–	–	1.83	●
37	T	0.80	○	14.66	○	0.54	◐	–	–	–	–	–	–
38	L	0.35	●	7.22	●	0.38	●	–	–	–	–	–	–
39	L	4.37	●	11.80	●	2.10	●	–	–	–	–	–	–
40	L	0.14	●	1071.52	●	0.03	○	1.26	●	–	–	18.39	●
41	L	0.44	●	11.00	●	0.03	○	–	–	–	–	–	–
42	L	0.71	●	15.09	●	0.77	●	–	–	–	–	–	–
43	L	2.59	●	8.00	●	2.26	●	2.96	●	–	–	–	–
44	N	0.29	○	6.76	○	0.31	○	17.43	○	572.51	○	–	–
45	T	0.28	○	9.62	○	0.02	○	–	–	–	–	0.55	●
46	L	0.28	●	7.31	●	0.29	●	–	–	–	–	0.19	●
47	L	0.14	●	7.77	●	0.09	●	1.37	●	–	–	40.43	●
48	T	0.24	○	8.44	○	0.02	○	–	–	–	–	–	–
49	T	0.24	○	7.72	○	0.03	○	0.62	○	–	–	–	–
50	T	0.23	○	8.18	○	0.03	○	0.66	○	–	–	1310.13	●
51	L	0.46	●	13.98	●	0.47	●	21.03	●	218.50	●	4.92	●
52	T	0.24	○	7.44	○	–	–	1.31	○	–	–	–	–
53	T	0.30	○	3.31	○	0.07	○	–	–	–	–	–	–
54	N	0.25	○	7.02	○	–	–	–	–	–	–	–	–
55	L	0.28	●	7.48	●	0.29	●	–	–	–	–	–	–
56	L	1.01	●	8.57	●	0.04	○	–	–	–	–	–	–
57	L	0.61	●	14.76	●	0.67	●	–	–	–	–	–	–
58	L	14.61	●	24.31	●	1.56	●	–	–	–	–	–	–
59	L	0.21	●	–	–	0.03	○	–	–	–	–	–	–
60	N	0.24	○	7.75	○	0.03	○	0.74	○	–	–	0.04	○
61	T	6.68	○	–	–	0.05	○	–	–	–	–	1.88	●

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

Table 3 Experimental results on 61 benchmarks drawn from Windows device drivers (runtime in seconds).

therefore presents a solution to the performance problems of QBF solvers referred to in Sections 5 and 6.

Bradley et al. [7] 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 [7] 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 [6]. 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. [3]. Their technique relies on the fact that some abstractions (of infinite-state systems) imply the existence of well-founded orders on the state space. Since neither one of these techniques leads to the explicit construction of 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.

Falke et al. [16] propose a sound abstraction of bit-vector programs to integer-based term rewriting systems. This sacrifices completeness, but enables the use of polynomial interpretations over (unbounded) integers in the rewriting system analyser, to the effect that non-linear ranking functions can be synthesised from some input programs.

8 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. A further opportunity for future work is termination analysis for low-level concurrent software with weak memory semantics, e.g., by means of instrumentation [1].

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. Furthermore, we are grateful for useful comments from Vijay D'Silva and Georg Weissenbacher, as well as the anonymous referees who identified important technical and presentational issues in previous revisions of this article.

References

1. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: European Symposium on Programming (ESOP), *Lecture Notes in Computer Science*, vol. 7792, pp. 512–532. Springer (2013)
2. Babic, D., Hu, A.J., Rakamaric, Z., Cook, B.: Proving termination by divergence. In: SEFM, pp. 93–102. IEEE (2007)
3. Ball, T., Kupferman, O., Sagiv, M.: Leaping loops in the presence of abstraction. In: CAV, *Lecture Notes in Computer Science*, vol. 4590, pp. 491–503. Springer (2007)
4. Benedetti, M.: sKizzo: A suite to evaluate and certify QBFs. In: CADE, *Lecture Notes in Computer Science*, vol. 3632, pp. 369–376. Springer (2005)
5. Biere, A.: Resolve and expand. In: SAT, *Lecture Notes in Computer Science*, vol. 3542, pp. 59–70. Springer (2005)
6. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: FMICS, *Electronic Notes in Theoretical Computer Science*, vol. 66, pp. 160–177. Elsevier (2002)
7. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination analysis of integer linear loops. In: CONCUR, *Lecture Notes in Computer Science*, vol. 3653, pp. 488–502. Springer (2005)
8. Brinkmann, R., Drechsler, R.: RTL-datapath verification using integer linear programming. In: Proc. of VLSI Design, pp. 741–746. IEEE (2002)
9. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Formal Methods in System Design **25**(2-3), 105–127 (2004)
10. Colón, M., Sipma, H.: Synthesis of linear ranking functions. In: TACAS, *Lecture Notes in Computer Science*, vol. 2031, pp. 67–81. Springer (2001)
11. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking function synthesis for bit-vector relations. In: TACAS, *Lecture Notes in Computer Science*, vol. 6015, pp. 236–250. Springer (2010)
12. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: SAS, *Lecture Notes in Computer Science*, vol. 3672, pp. 87–101. Springer (2005)
13. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426. ACM (2006)
14. Dams, D., Gerth, R., Grumberg, O.: A heuristic for the automatic generation of ranking functions. In: Workshop on Advances in Verification, pp. 1–8 (2000)
15. Encrenaz, E., Finkel, A.: Automatic verification of counter systems with ranking functions. In: INFINITY, *Electronic Notes in Theoretical Computer Science*, vol. 239, pp. 85–103. Elsevier (2009)
16. Falke, S., Kapur, D., Sinz, C.: Termination analysis of imperative programs using bitvector arithmetic. In: VSTTE, *Lecture Notes in Computer Science*, vol. 7152, pp. 261–277. Springer (2012)
17. Giunchiglia, E., Narizzano, M., Tacchella, A.: QuBE++: an efficient QBF solver. In: FMCAD, *Lecture Notes in Computer Science*, vol. 3312, pp. 201–213. Springer (2004)
18. Griggio, A.: Effective word-level interpolation for software verification. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 28–36. IEEE (2011)
19. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI, pp. 35–46. ACM (1988)

20. Jussila, T., Biere, A.: Compressing BMC encodings with QBF. In: Workshop on Bounded Model Checking (BMC'06), *Electronic Notes in Theoretical Computer Science*, vol. 174, pp. 45–56. Elsevier (2007)
21. Jussila, T., Biere, A., Sinz, C., Kroening, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: SAT, *Lecture Notes in Computer Science*, vol. 4501, pp. 201–214. Springer (2007)
22. Kovásznai, G., Fröhlich, A., Biere, A.: On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In: SMT Workshop at IJCAR (2012)
23. Parthasarathy, G., Iyer, M.K., Cheng, K.T., Wang, L.C.: An efficient finite-domain constraint solver for circuits. In: Design Automation Conference (DAC), pp. 212–217. ACM (2004)
24. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI, *Lecture Notes in Computer Science*, vol. 2937, pp. 239–251. Springer (2004)
25. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE (2004)
26. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: PADL, *Lecture Notes in Computer Science*, vol. 4354, pp. 245–259. Springer (2007)
27. 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)
28. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: LPAR, *Lecture Notes in Computer Science*, vol. 5330, pp. 274–289. Springer (2008)
29. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
30. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time (preliminary report). In: STOC, pp. 1–9. ACM (1973)
31. Wegner, P.: A technique for counting ones in a binary computer. Commun. ACM **3**(5), 322 (1960)
32. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: Efficiently solving quantified bit-vector formulas. Formal Methods in System Design **42**, 3–23 (2013)
33. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: CAV, *Lecture Notes in Computer Science*, vol. 5123, pp. 385–398. Springer (2008)