Termination Analysis of Imperative Programs Using Bitvector Arithmetic*

Stephan Falke[1], Deepak Kapur[2], and Carsten Sinz[1]

[1] Institute for Theoretical Computer Science, KIT, Germany
{stephan.falke,carsten.sinz}@kit.edu
[2] Dept. of Computer Science, University of New Mexico, USA
kapur@cs.unm.edu

Abstract. Currently, nearly all methods for proving termination of imperative programs apply an unsound and incomplete abstraction by treating bitvectors and bitvector arithmetic as (unbounded) integers and integer arithmetic, respectively. This abstraction ignores the wrap-around behavior caused by under- and overflows in bitvector arithmetic operations. This is particularly problematic in the termination analysis of low-level system code. This paper proposes a novel method for encoding the wrap-around behavior of bitvector arithmetic within integer arithmetic. Afterwards, existing methods for reasoning about the termination of integer arithmetic programs can be employed for reasoning about the termination of bitvector arithmetic programs. An empirical evaluation shows the practicality and effectiveness of the proposed method.

1 Introduction

Most methods for proving termination of imperative programs (e.g., [1,3,4,5,6,7,9,10,14,18,19,20,23,24] and including our own recent work [13]) developed during the past decade deviate in their analysis in one important aspect from the execution of a program on a computer: machine arithmetic operating on bitvectors of a limited range is treated as arithmetic on (unbounded) integers (or as arithmetic on real numbers). Thus, the wrap-around behavior caused by under- and overflow is ignored and the semantics of the computer program is only approximated.

This approximation is undesirable and can err in both directions:

- A program may be terminating using bitvector arithmetic, but nonterminating using integer arithmetic.
- A program may be terminating using integer arithmetic, but nonterminating using bitvector arithmetic.

Example 1. Consider the following C functions:

* This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

R. Joshi, P. Müller, and A. Podelski (Eds.): VSTTE 2012, LNCS 7152, pp. 261–277, 2012.

```
void f(int i)
{
    while (i > 0) {
        ++i;
    }
}
```

```
void g(int i, int j)
{
    while (i <= j) {
        ++i;
    }
}
```

Then `f` is terminating using bitvector arithmetic since `i` will eventually overflow and become negative.[1] On the other hand, `f` is nonterminating using integer arithmetic since the loop-condition stays always true whenever the argument passed to `f` is at least 1. For `g` the situation is reversed. It is nonterminating using bitvector arithmetic if `j` is `INT_MAX`, but terminating using integer arithmetic since `i` will eventually exceed `j`. ◊

This paper presents an adaptation of the method developed in [13] that correctly models the wrap-around behavior of bitvector arithmetic. For this, bitvectors are represented by integers and the wrap-around behavior is explicitly modeled in the integer domain using a normalization step after each arithmetic operation. The technique is not specific to [13], however, and can be combined with other methods for the termination analysis of imperative programs.

The only previous work concerned with the termination analysis of programs using bitvector arithmetic is, to the best of our knowledge, [8]. That paper has developed two methods for the synthesis of ranking functions for programs using bitvector arithmetic:

1. An encoding of bitvector arithmetic within integer arithmetic. This is similar in spirit to our approach, but [8] requires the use of quantifiers. These quantifiers need to be eliminated before the ranking functions can be synthesized. Quantifier elimination is an expensive operation in general. In contrast to this, the approach developed in the present paper does not introduce any quantifiers and thus does not rely on quantifier elimination.
2. An approach based on template matching for linear ranking functions. This approach uses a SAT, QBF, or (in the later [26]) SMT solver in order to instantiate the templates by solving quantified bitvector formulas of the shape $\exists x_1, \ldots, x_n.\ \forall y_1, \ldots, y_m.\ \psi$. While QBF solvers can directly be applied to such formulas, they currently lack in performance and are not very successful in solving the generated formulas. The SMT solver from [26] performs better than current QBF solvers but its performance is still not completely satisfactory. In order to apply SAT solvers to quantified bitvector formulas of the shape $\exists x_1, \ldots, x_n.\ \forall y_1, \ldots, y_m.\ \psi$, [8] uses the following approach: the

[1] Strictly speaking, a signed under- or overflow yields undefined behavior according to the C99 standard. Virtually all compilers treat signed under- and overflows using the wrap-around behavior, though. This is also the required behavior for unsigned under- and overflows according to the C99 standard. This paper assumes the wrap-around behavior semantics for both signed and unsigned under- and overflows.

existentially quantified variables $x_1, \ldots, x_n$ are instantiated by the bitvectors corresponding to values from $\{-1, 0, 1\}$, and for each of these possibilities, the instantiated quantifier-free formula $\neg\psi$ is checked for unsatisfiability. Using this approach, the SAT-based implementation of [8] is quite efficient but fails to generate ranking functions such as $2x - y$ which can easily be generated by our implementation.

An empirical comparison of our method as implemented in the tool KITTeL [13] with the methods from [8] (and [26]) is very encouraging. Of the 61 examples considered in [8] (51 terminating ones and 10 nonterminating ones), our implementation succeeds in proving termination of 38 examples. The methods from [8] (and [26]) succeed on 30 examples (quantifier-elimination-based approach), 34 examples (template-based approach with a SAT solver), 8 examples (template-based approach with a QBF solver), and 27 examples (template-based approach with the SMT solver from [26]), respectively.

The problem of unsoundness if bitvectors are abstracted to integers is also present in other static program analysis methods. Similar to termination analysis, this problem is rarely addressed, but invariant generation methods based on modular arithmetic are presented in [17,21]. Furthermore, decision procedures for modular arithmetic have been developed in [2,25].

This paper is organized as follows: Sect. 2 briefly introduces LLVM [15] since our method makes use of this compiler framework. Then, Sect. 3 recalls the method for termination analysis developed in [13], but presents it in a new light. Next, Sect. 4 shows how the wrap-around behavior of bitvector arithmetic can be modeled by introducing explicit normalization steps. Section 5 presents an empirical comparison with the methods of [8] (and [26]), and Sect. 6 concludes.

2 A Brief Overview of LLVM and Its Use in KITTeL

Termination analysis of programs written in real-life programming languages is a very important, yet notoriously difficult, task. This is in part due to the rich syntax of these languages. Furthermore, their complex and sometimes ambiguous semantics gives rise to further intricacies. The tool KITTeL [13] thus performs the termination analysis not on the source code level but on the level of a compiler intermediate representation (IR). This approach has the following advantages:

1. The IR is considerably simpler than real-life programming languages. This makes it possible to accept arbitrary (valid) programs as input.
2. The program whose termination behavior is analyzed is much closer to the program that is actually executed on the computer since most ambiguities of the semantics have already been resolved.
3. It becomes possible to analyze the termination behavior of programs written in any programming language that can be converted into the IR by a compiler front-end.

More concretely, KITTeL is based on the LLVM compiler framework and its intermediate language LLVM-IR [15]. Since there are compilers for various programming languages built atop of LLVM, KITTeL can be used for the termination

analysis of programs written in C, C++, Objective-C, Ada, Fortran, etc. The main goal of KITTeL is the termination analysis of C programs.

A C program is first compiled into an LLVM-IR program using existing compiler front-ends such as `llvm-gcc` or `clang`. The LLVM-IR program is next converted into a transition system. This transition system is represented in the form of an `int`-*based term rewrite system (*`int`-*based TRS)* [12,13] in KITTeL, and the termination analysis itself is performed on this `int`-based TRS.[2] Fig. 1 gives an overview of the approach.

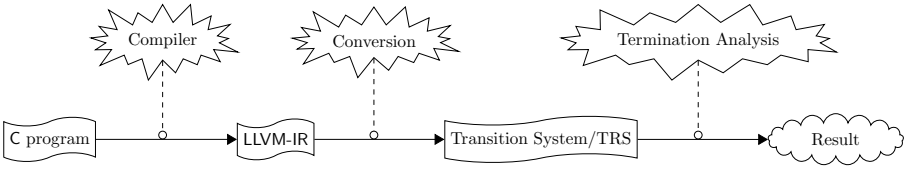


Fig. 1. Bird's-eye view of KITTeL's approach

2.1 LLVM-IR

An LLVM-IR program is an assembly program for a register machine with an unbounded number of registers. All registers in LLVM-IR are typed. Available types include a void type, integer types like `i32` (where the bit-width is given explicitly but signed and unsigned types are not distinguished), floating point types, and derived types (such as pointer, array and structure types). The type `i1` serves as a Boolean type. An LLVM-IR program consists of one or more functions. Each function is given as a list of basic blocks, where each basic block is a list of instructions. Execution of a function starts at the first basic block in the list.

LLVM-IR instructions can roughly be categorized into six classes:

1. *Three-address code (TAC)* instructions for arithmetical operations, comparison operations, and bitwise operations.
2. *Control flow* instructions: conditional and unconditional *branch* (`br`) instructions, *return* (`ret`) instructions, and *phi* (`phi`) instructions.
3. *Function calls* using `call` instructions.
4. *Memory access* instructions, namely `load` and `store` instructions.
5. *Address calculations* using `getelementptr` instructions.
6. *Auxiliary instructions* like type cast instructions (which do not change the bitlevel representation), signed and unsigned extension instructions, and truncation instructions.

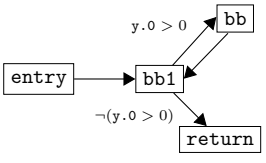
Branch instructions and return instructions only occur as the last instruction of a basic block and each basic block is terminated by one of these instructions.

[2] Alternatively, it would be possible to apply methods based on ranking functions [3,4,6,7,19], possibly combined with abstraction refinement [9,10,14,20], in order to investigate the termination behavior of the transition system.

A function gives rise to a *basic block graph* [22] which contains one node for each basic block and an edge from the basic block *bb* to the basic block *bb'* if *bb* is terminated by a branch instruction that can branch to *bb'*. For conditional branches, this edge is labeled by the condition under which the branch is taken.

Example 2. The following figure shows a C program, the LLVM-IR program obtained using the compiler front-end `llvm-gcc`, and the basic block graph of the function `power`:

```
int power(int x, int y) {
    int r = 1;
    while (y > 0) {
        r = r*x;
        y = y - 1;
    }
    return r;
}
```




```
define i32 @power(i32 %x, i32 %y) {
entry:
  br label %bb1

bb1:
  %y.0 = phi i32 [ %y, %entry ], [ %2, %bb ]
  %r.0 = phi i32 [ 1, %entry ], [ %1, %bb ]
  %0 = icmp sgt i32 %y.0, 0
  br i1 %0, label %bb, label %return

bb:
  %1 = mul i32 %r.0, %x
  %2 = sub i32 %y.0, 1
  br label %bb1

return:
  ret i32 %r.0
}
```

This example is used in the next section in order to illustrate the conversion of an LLVM-IR program into a transition system/`int`-based TRS. ◊

LLVM-IR programs are in *static single assignment (SSA)* form, i.e., each register (variable) is assigned exactly once in the static program. This requires the use of phi-instructions, which may only occur at the beginning of basic blocks and select one of several values whenever the control flow in a program converges (e.g., after an `if-then-else` statement). Thus, the meaning of `%r.0 = phi i32 [ 1, %entry ], [ %1, %bb ]` contained in the basic block `bb1` in Exa. 2 is that the register `%r.0` is assigned the value `1` if the control flow passed from `entry` to `bb1` and the value contained in `%1` if the control passed from `bb` to `bb1`.

3 Termination Analysis Using LLVM

For the termination analysis using LLVM, an LLVM-IR program is converted into a transition system whose termination behavior is then investigated. Within `KITTeL`, the transition system is represented in the form of an `int`-based TRS.

3.1 Translating LLVM-IR Programs into Transition Systems

For ease of exposition, it is assumed that the LLVM-IR program operates only on integer types, that there is exactly one function, and that this function does not

contain any function calls. It thus only contains arithmetical and bitwise TAC instructions, comparison instructions, control flow instructions, and auxiliary instructions.[3]

For the translation into a transition system, each integer-typed function argument (i.e., of a type `i`k with $k > 1$) and each register defined by an integer-typed TAC instruction or phi-instruction is mapped to a variable. The set of these variables is denoted by $\mathcal{V}$. The transition system $(S, \Lambda, \rightarrow)$ corresponding to the LLVM-IR program is now constructed as follows:

- S contains one element for each node in the basic block graph.
- Λ denotes the set of transition constraints, which are quantifier-free constraints from (non-linear) integer arithmetic over the variables $\mathcal{V}$ and their primed versions in $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. As usual, the intended semantics of $\mathcal{V}'$ is to refer to the values of the variables in $\mathcal{V}$ after executing a transition.
- $\longrightarrow \subseteq S \times \Lambda \times S$ denotes the set of transitions: for each branch from the basic block *bb* to the basic block *bb'*, let $bb \xrightarrow{\lambda} bb'$ such that λ
  - describes the effect of the integer-typed TAC instructions in *bb*,
  - contains the condition under which the branch is taken, and
  - instantiates the variables corresponding to integer-typed phi-instructions in *bb'* according to their value if the control flow passes from *bb* to *bb'*.

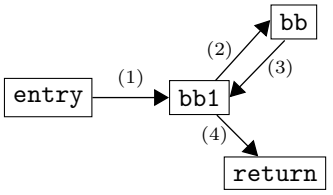
The transition system is thus the basic block graph where each edge is, in addition to the branch condition, labeled by the state change caused by the transition.

In [13], bitvectors and bitvector arithmetic are abstracted to integers and integer arithmetic, respectively. Furthermore, there is no distinction between signed and unsigned operations. The arithmetical TAC instructions `add`, `sub`, and `mul` are replaced by the obvious integer arithmetic operations. The effect of the TAC instructions `sdiv`, `udiv`, `srem`, and `urem` is not modeled exactly. Instead, their result is abstracted to a fresh variable (optionally, constraints can restrict the range of this fresh variable, see [13] for details). The bitwise TAC instructions `and`, `or`, and `xor` are handled the same way. Finally, the extension and truncation instructions `sext`, `uext`, and `trunc` are modeled to not change the value of the integer.

Comparison instructions used in branch conditions are replaced in the obvious way by the corresponding integer comparisons. Signed and unsigned comparisons are not distinguished, i.e., `icmp ugt` and `icmp sgt` are both replaced by $>$. Boolean-typed TAC instructions used in branch conditions (`and`, `or`, and `xor`) are converted in the obvious way as well.

Example 3. For the LLVM-IR program from Exa. 2, the following transition system is obtained:

[3] Multiple (recursive) functions are handled by abstracting the return value of the called function by a fresh variable and by introducing suitable transitions to the entry state of the called function. Since `KITTeL` currently does not model the memory content, pointers are not tracked, `store` instructions are handled as no-ops, and `load` instructions are abstracted to fresh variables. Similarly, floating point instructions are handled as no-ops or abstracted to fresh variables.



(1) $\mathtt{y.0}' \simeq \mathtt{y} \wedge \mathtt{r.0}' \simeq 1$
(2) $\mathtt{y.0} > 0$
(3) $\mathtt{y.0}' \simeq \mathtt{y.0} - 1 \wedge \mathtt{r.0}' \simeq \mathtt{r.0} * \mathtt{x} \wedge$
$\mathtt{\%1}' \simeq \mathtt{r.0} * \mathtt{x} \wedge \mathtt{\%2}' \simeq \mathtt{y.0} - 1$
(4) $\neg(\mathtt{y.0} > 0)$

Here, $\mathcal{V} = \{\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \mathtt{\%1}, \mathtt{\%2}\}$. ◊

3.2 Representing Transition Systems Using int-Based TRSs

A transition system can be represented in the form of an `int`-based TRS [12,13]. The rewrite rules of an `int`-based TRS have the form $f(x_1, \ldots, x_n) \to g(p_1, \ldots, p_m)\ [\![\varphi]\!]$ where f and g are (uninterpreted) function symbols, $x_1, \ldots, x_n$ are pairwise distinct variables, $p_1, \ldots, p_m$ are (possibly non-linear) polynomials, and φ is a quantifier-free constraint from (non-linear) integer arithmetic that guards when a rewrite rule can be applied.[4] Then, each transition $s_1 \xrightarrow{\lambda} s_2$ gives rise to a rewrite rule

$$s_1(x_1, \ldots, x_n) \to s_2(e_1, \ldots, e_n)\ [\![\varphi]\!]$$

where $x_1, \ldots, x_n$ is a fixed order of the variables in $\mathcal{V}$ and

- $e_i = p$ if λ contains an "assignment" $x_i' \simeq p$ and $e_i = x_i$ otherwise.
- φ is λ with the "assignments" removed.

Example 4. Continuing Exa. 3,

$$\begin{aligned}
\mathtt{entry}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \mathtt{\%1}, \mathtt{\%2}) &\to \mathtt{bb1}(\mathtt{x}, \mathtt{y}, \mathtt{y}, 1, \mathtt{\%1}, \mathtt{\%2}) \\
\mathtt{bb1}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \mathtt{\%1}, \mathtt{\%2}) &\to \mathtt{bb}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \mathtt{\%1}, \mathtt{\%2})\ [\![\mathtt{y.0} > 0]\!] \\
\mathtt{bb1}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \mathtt{\%1}, \mathtt{\%2}) &\to \mathtt{return}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \mathtt{\%1}, \mathtt{\%2})\ [\![\neg(\mathtt{y.0} > 0)]\!] \\
\mathtt{bb}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \mathtt{\%1}, \mathtt{\%2}) &\to \mathtt{bb1}(\mathtt{x}, \mathtt{y}, \mathtt{y.0} - 1, \mathtt{r.0}, \mathtt{r.0} * \mathtt{x}, \mathtt{y.0} - 1)
\end{aligned}$$

is the representation of the transition system as an `int`-based TRS. ◊

When we talk about adding rewrite rules or replacing rewrite rules in the following, this can equally well be thought of as adding transitions or replacing transitions in the transition system (possibly adding new states as well if the rewrite rules introduce new function symbols).

The translation as outlined above produces rewrite rules where the function symbols have an unnecessarily large number of arguments. The number of arguments can be reduced by adapting standard compiler techniques:

1. *Backward slicing* with respect to the constraints removes all variables that are not relevant for the control flow of the program. In the running example, this removes the arguments corresponding to `x`, `r.0`, `%1`, and `%2`.

[4] In contrast to ordinary TRSs, $p_1, \ldots, p_m$ and φ may contain variables not occurring in $x_1, \ldots, x_n$.

2. *Liveness analysis* removes variables before they are defined or after they are no longer needed. In the running examples, this removes the argument corresponding to `y.0` from the function symbols `entry` and `return` and the argument corresponding to `y` from all function symbols but `entry`.

Example 5. Applying these methods,

$$\texttt{entry(y)} \to \texttt{bb1(y)} \tag{1}$$
$$\texttt{bb1(y.0)} \to \texttt{bb(y.0)} \ [\![\texttt{y.0} > 0]\!] \tag{2}$$
$$\texttt{bb1(y.0)} \to \texttt{return()} \ [\![\neg(\texttt{y.0} > 0)]\!] \tag{3}$$
$$\texttt{bb(y.0)} \to \texttt{bb1(y.0} - 1) \tag{4}$$

is the final `int`-based TRS obtained from Exa. 4. ◊

3.3 Termination Analysis of `int`-Based TRSs

The `int`-based TRS obtained from an LLVM-IR program is then analyzed for termination using term rewriting techniques. The key techniques are:

1. Determining which rules may be applied following each other and a decomposition into non-trivial strongly connected components (SCCs). This step roughly corresponds to a decomposition into loops of the program.
2. Automatically generating well-founded relations based on (possibly non-linear) polynomial interpretations. Here, a polynomial interpretation maps the function symbols to polynomials such that a function symbol f with n arguments is mapped to a polynomial $\mathcal{P}ol(f) \in \mathbb{Z}[X_1, \ldots, X_n]$. A polynomial interpretation is applied to terms by applying the polynomial assigned to the function symbol to the arguments, i.e., by letting $\mathcal{P}ol(f(p_1, \ldots, p_n)) = \mathcal{P}ol(f)(p_1, \ldots, p_n)$. A polynomial interpretation gives rise to a well-founded relation by letting (for terms s, t and a quantifier-free integer constraint φ)

$$s \succ_{\mathcal{P}ol} t \ [\![\varphi]\!] \quad \text{iff} \quad \varphi \Rightarrow \mathcal{P}ol(s) > \mathcal{P}ol(t) \text{ and } \varphi \Rightarrow \mathcal{P}ol(s) \geq 0 \text{ are valid}$$

   Similarly, $s \succsim_{\mathcal{P}ol} t \ [\![\varphi]\!]$ iff $\varphi \Rightarrow \mathcal{P}ol(s) \geq \mathcal{P}ol(t)$ is valid (these relations are in general undecidable due to non-linearity). Then, all rules $s \to t \ [\![\varphi]\!]$ with $s \succ_{\mathcal{P}ol} t \ [\![\varphi]\!]$ can be removed from an `int`-based TRS if all remaining rules $s' \to t' \ [\![\varphi']\!]$ satisfy $s' \succsim_{\mathcal{P}ol} t' \ [\![\varphi']\!]$. Methods for the automatic generation of suitable polynomial interpretations are discussed in [12,13]. The current implementation is restricted to polynomials of the form $\sum_{i=1}^{n} a_i X_i + \sum_{j=1}^{n} b_j X_j^2 + c$, with $a_i, b_j, c \in \mathbb{Z}$. Notice that this is more general than the linear ranking functions that are considered in [8] since the coefficients are not restricted and a limited form of non-linearity is supported.
3. Combining rewrite rules that may be applied following each other into new rewrite rules. This corresponds to combining transitions that may be applied successively in the transition system.

These techniques can be applied modularly in any order and combination. Concretely, `KITTeL` applies them in a loop, where each loop iteration applies the first

technique from the list that is successfully applicable. For SCC decomposition and the generation of polynomial interpretations, SMT-solvers such as `Yices` [11] or `Z3` [16] are used for solving (linear) integer arithmetic constraints, where `KITTeL` uses `Yices` by default. Details are discussed in [12,13].

Example 6. Termination of the `int`-based TRS from Exa. 5 is easily established using these techniques, thus establishing termination of the C program using integer arithmetic. First, SCC decomposition removes the rewrite rules (1) and (3). Next, the polynomial interpretation $\mathcal{P}ol(\texttt{bb1}) = X_1$, $\mathcal{P}ol(\texttt{bb}) = X_1 - 1$ removes the rewrite rule (2) since $\texttt{bb1(y.0)} \succ_{\mathcal{P}ol} \texttt{bb(y.0)}\ [\![\texttt{y.0} > 0]\!]$ and $\texttt{bb(y.0)} \succsim_{\mathcal{P}ol} \texttt{bb1(y.0} - 1)$. Finally, SCC decomposition finishes the termination proof. ◊

4 Encoding Bitvector Arithmetic

In order to model the bitvector semantics of machine arithmetic, bitvectors can be represented by (unbounded) integers if the wrap-around behavior of the arithmetical operations is modeled properly. There are two natural choices for the representation of a bitvector by an integer:

1. The bitvector $b_{n-1} \cdots b_0$ is represented by its *unsigned* value $\sum_{i=0}^{n-1} b_i \cdot 2^i \in \{0, \ldots, 2^n - 1\}$.
2. The bitvector $b_{n-1} \cdots b_0$ is represented by its *signed* (two's complement) value $-b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$.

In the following, the representation by the signed value is considered (the implementation in `KITTeL` supports both possibilities).

The semantics of the bitvector operations and the wrap-around behavior of bitvector arithmetic is handled by a two-phase approach:

1. For the generation of the transition system/`int`-based TRS, the conversion of comparison instructions, arithmetical and bitwise TAC instructions, and extension/truncation instructions is adapted in order to correspond to their semantics on bitvectors. In particular, signed and unsigned operations are carefully distinguished. The wrap-around behavior of the arithmetical operations is not yet taken into account since this will be done afterwards in the second phase.
2. The wrap-around behavior of the arithmetical instructions is modeled by explicitly normalizing the results of arithmetical operations in order to ensure that they are within the appropriate ranges.

In the following, $\|x\|$ for a variable x denotes the size of the bitvector of the LLVM-IR value corresponding to x. This extends to arithmetical expressions in the obvious way. Furthermore, $\mathsf{intmin}(n) = -2^{n-1}$ and $\mathsf{intmax}(n) = 2^{n-1} - 1$ denote the minimal and maximal signed value that can be represented by bitvectors of size n.

4.1 Phase 1

In contrast to Sect. 3, signed and unsigned comparison instructions are now distinguished. Since a bitvector is represented by its signed value, the signed comparison instructions are still converted to the corresponding integer comparisons and only the unsigned comparison instructions need to be handled differently.

`icmp eq i`k x`, `y	$x \simeq y$	`icmp ne i`k x`, `y	$x \not\simeq y$
`icmp ugt i`k x`, `y	$\mathsf{ugt}(x, y)$	`icmp sgt i`k x`, `y	$x > y$
`icmp uge i`k x`, `y	$\mathsf{ugt}(x, y) \vee x \simeq y$	`icmp sge i`k x`, `y	$x \geq y$
`icmp ult i`k x`, `y	$\mathsf{ult}(x, y)$	`icmp slt i`k x`, `y	$x < y$
`icmp ule i`k x`, `y	$\mathsf{ult}(x, y) \vee x \simeq y$	`icmp sle i`k x`, `y	$x < y$

Here, $\mathsf{ugt}(x, y)$ and $\mathsf{ult}(x, y)$ encode unsigned greater-than and less-than comparisons on bitvectors, respectively, where bitvectors are represented by their signed value. They are defined as follows:

$$\begin{aligned}\mathsf{ugt}(x, y) = \quad & (x \geq 0 \wedge y \geq 0 \wedge x > y) \\ \vee\ & (x \geq 0 \wedge y < 0) \\ \vee\ & (x < 0 \wedge y < 0 \wedge x > y) \\ \mathsf{ult}(x, y) = \ & \mathsf{ugt}(y, x)\end{aligned}$$

Arithmetical and bitwise TAC instructions as well as auxiliary instructions are converted as follows. Again, signed and unsigned operations are carefully distinguished. The results of `sdiv`, `udiv`, `srem`, `urem`, `and`, and `or` instructions are represented by a fresh variable n_i. Furthermore, constraints on the value of this variable are added. These constraints describe the (approximated) semantics of the corresponding instructions on bitvectors.

arithmetic	$z =$ `add i`k x`, `y	$z' \simeq x + y$
	$z =$ `sub i`k x`, `y	$z' \simeq x - y$
	$z =$ `mul i`k x`, `y	$z' \simeq x * y$
	$z =$ `sdiv i`k x`, `y	$z' \simeq n_i \wedge \mathsf{sdiv}(x, y, n_i)$
	$z =$ `udiv i`k x`, `y	$z' \simeq n_i \wedge \mathsf{udiv}(x, y, n_i)$
	$z =$ `srem i`k x`, `y	$z' \simeq n_i \wedge \mathsf{srem}(x, y, n_i)$
	$z =$ `urem i`k x`, `y	$z' \simeq n_i \wedge \mathsf{urem}(x, y, n_i)$
bitwise operations	$z =$ `and i`k x`, `y	$z' \simeq n_i \wedge \mathsf{and}(x, y, n_i)$
	$z =$ `or i`k x`, `y	$z' \simeq n_i \wedge \mathsf{or}(x, y, n_i)$
	$z =$ `xor i`k x`, `y	$z' \simeq n_i$
extension/truncation	$z =$ `sext i`k x `to i`l	$z' \simeq n_i$
	$z =$ `uext i`k x `to i`l	$z' \simeq x \wedge \mathsf{uext}(z, n_i)$
	$z =$ `trunc i`k x `to i`l	$z' \simeq x$

The formulas for the constraints typically perform a case distinction on the inputs x and y in order to obtain constraints on the result n_i. For instance for `and`, if both x and y are positive (most significant bit is zero), then n_i is positive as well and exceeds neither x nor y. If exactly one of x and y is positive, then

n_i is positive and does not exceed the positive input. Finally, if both x and y are negative (most significant bit is one), then n_i is negative as well and again exceeds neither x nor y. The formulas are as follows:

$$\begin{aligned}
\mathsf{sdiv}(x,y,n_i) = \quad & (x \simeq 0 \wedge n_i \simeq 0) \vee (y \simeq 1 \wedge n_i \simeq x) \vee (y \simeq -1 \wedge n_i \simeq -x) \\
& \vee (y > 1 \wedge x > 0 \wedge n_i \geq 0 \wedge n_i < x) \\
& \vee (y > 1 \wedge x < 0 \wedge n_i \leq 0 \wedge n_i > x) \\
& \vee (y < -1 \wedge x > 0 \wedge n_i \leq 0 \wedge n_i > -x) \\
& \vee (y < -1 \wedge x < 0 \wedge n_i \geq 0 \wedge n_i < -x) \\
\mathsf{udiv}(x,y,n_i) = \quad & (x \simeq 0 \wedge n_i \simeq 0) \vee (y \simeq 1 \wedge n_i \simeq x) \\
& \vee (\mathsf{ugt}(y,1) \wedge \mathsf{ugt}(x,0) \wedge \mathsf{ugt}(n_i,0) \wedge \mathsf{ult}(n_i,x)) \\
\mathsf{srem}(x,y,n_i) = \quad & (x \simeq 0 \wedge n_i \simeq 0) \vee (y \simeq 1 \wedge n_i \simeq 0) \vee (y \simeq -1 \wedge n_i \simeq 0) \\
& \vee (y > 1 \wedge x > 0 \wedge n_i \geq 0 \wedge n_i < y) \\
& \vee (y > 1 \wedge x < 0 \wedge n_i \leq 0 \wedge n_i > -y) \\
& \vee (y < -1 \wedge x > 0 \wedge n_i \geq 0 \wedge n_i < y) \\
& \vee (y < -1 \wedge x < 0 \wedge n_i \leq 0 \wedge n_i > y) \\
\mathsf{urem}(x,y,n_i) = \quad & (x \simeq 0 \wedge n_i \simeq 0) \vee (y \simeq 1 \wedge n_i \simeq 0) \\
& \vee (\mathsf{ugt}(y,1) \wedge \mathsf{ugt}(x,0) \wedge \mathsf{ult}(n_i,y)) \\
\mathsf{and}(x,y,n_i) = \quad & (x \geq 0 \wedge y \geq 0 \wedge n_i \geq 0 \wedge n_i \leq x \wedge n_i \leq y) \\
& \vee (x \geq 0 \wedge y < 0 \wedge n_i \geq 0 \wedge n_i \leq x) \\
& \vee (x < 0 \wedge y \geq 0 \wedge n_i \geq 0 \wedge n_i \leq y) \\
& \vee (x < 0 \wedge y < 0 \wedge n_i < 0 \wedge n_i \leq x \wedge n_i \leq y) \\
\mathsf{or}(x,y,n_i) = \quad & (x \geq 0 \wedge y \geq 0 \wedge n_i \geq 0 \wedge n_i \geq x \wedge n_i \geq y) \\
& \vee (x \geq 0 \wedge y < 0 \wedge n_i < 0 \wedge n_i \geq y) \\
& \vee (x < 0 \wedge y \geq 0 \wedge n_i < 0 \wedge n_i \geq x) \\
& \vee (x < 0 \wedge y < 0 \wedge n_i < 0 \wedge n_i \geq x \wedge n_i \geq y) \\
\mathsf{uext}(x,n_i) = \quad & (x \geq 0 \wedge n_i \simeq x) \vee \left(x < 0 \wedge n_i \simeq x + 2^{\|x\|}\right)
\end{aligned}$$

Example 7. Recall the function `f` from Exa. 1:

```
void f(int i) {
    while (i > 0) {
        ++i;
    }
}
```

```
define void @f(i32 %i) {
entry:
  br label %bb1

  bb1:
    %i.0 = phi i32 [ %i, %entry ], [ %1, %bb ]
    %0 = icmp sgt i32 %i.0, 0
    br i1 %0, label %bb, label %return

  bb:
    %1 = add nsw i32 %i.0, 1
    br label %bb1

  return:
    ret void
}
```

$$\begin{aligned}
&\texttt{entry}(\texttt{i}) \to \texttt{bb1}(\texttt{i}) \\
&\texttt{bb1}(\texttt{i.0}) \to \texttt{bb}(\texttt{i.0}) \; [\![\texttt{i.0} > 0]\!] \\
&\texttt{bb1}(\texttt{i.0}) \to \texttt{return}() \; [\![\neg(\texttt{i.0} > 0)]\!] \\
&\texttt{bb}(\texttt{i.0}) \to \texttt{bb1}(\texttt{i.0} + 1)
\end{aligned}$$

After phase 1 and the slicing/liveness analysis based elimination of arguments, the `int`-based TRS show above is obtained. Notice that the wrap-around behavior caused by an overflow in $\texttt{i.0} + 1$ it not yet taken care of. This is addressed in phase 2. ◊

4.2 Phase 2

Next, the wrap-around behavior of the arithmetical instructions is modeled by modifying the generated `int`-based TRS. For this, the wrap-around behavior is simulated by explicitly normalizing the resulting integers to be in the appropriate ranges. For a variable x, the normalization consists of the repeated addition or subtraction of the correction $2^{\|x\|}$ until x is in the range of bitvectors of size $\|x\|$. The following construction is applied separately to each rewrite rule.

No arithmetical operations in the constraint: At first, it is assumed that the constraint of the rewrite rule does not contain any arithmetical operations. The general case is discussed subsequently.

Then, the rewrite rule

$$\rho : f(x_1, \ldots, x_n) \to g(p_1, \ldots, p_m) \ [\![\varphi]\!]$$

is replaced by the following rewrite rules (notice that $p_1, \ldots, p_m$ may be outside of the appropriate ranges even if the instantiations of all variables are within their appropriate ranges):

$$\begin{aligned}
f(x_1, \ldots, x_n) &\to g^\sharp(p_1, \ldots, p_m) \ [\![\varphi \wedge \mathsf{inrange}(\mathcal{V}(\rho))]\!] \\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1 + 2^{\|x_1\|}, \ldots, x_m) \ [\![x_1 < \mathsf{intmin}(\|x_1\|)]\!] \\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1 - 2^{\|x_1\|}, \ldots, x_m) \ [\![x_1 > \mathsf{intmax}(\|x_1\|)]\!] \\
&\vdots \\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1, \ldots, x_m + 2^{\|x_m\|}) \ [\![x_m < \mathsf{intmin}(\|x_m\|)]\!] \\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1, \ldots, x_m - 2^{\|x_m\|}) \ [\![x_m > \mathsf{intmax}(\|x_m\|)]\!] \\
g^\sharp(x_1, \ldots, x_m) &\to g(x_1, \ldots, x_m) \ [\![\mathsf{inrange}(\{x_1, \ldots, x_m\})]\!]
\end{aligned}$$

Here, $g^\sharp$ is a fresh function symbol, $\mathcal{V}(\rho)$ are the variables occurring in ρ, and

$$\mathsf{inrange}(V) = \bigwedge_{v \in V} (v \geq \mathsf{intmin}(\|v\|) \wedge v \leq \mathsf{intmax}(\|v\|))$$

expresses that all arithmetical expressions in V are in the appropriate ranges.[5]

[5] It of course suffices to add the $g^\sharp(\ldots) \to g^\sharp(\ldots)$ $[\![\ldots]\!]$ rules only for those x_i where p_i contains arithmetical operations. If there is no such i, then the rewrite rule ρ can be taken as is.

Example 8. Continuing Exa. 7, phase 2 produces

$$\texttt{entry(i)} \to \texttt{bb1(i)}\ [\![\mathsf{inrange}(\{\texttt{i}\})]\!] \tag{5}$$
$$\texttt{bb1(i.0)} \to \texttt{bb(i.0)}\ [\![\texttt{i.0} > 0 \wedge \mathsf{inrange}(\{\texttt{i.0}\})]\!] \tag{6}$$
$$\texttt{bb1(i.0)} \to \texttt{return()}\ [\![\neg(\texttt{i.0} > 0) \wedge \mathsf{inrange}(\{\texttt{i.0}\})]\!] \tag{7}$$
$$\texttt{bb(i.0)} \to \texttt{bb1}^\sharp\texttt{(i.0} + 1)\ [\![\mathsf{inrange}(\{\texttt{i.0}\})]\!] \tag{8}$$
$$\texttt{bb1}^\sharp\texttt{(i.0)} \to \texttt{bb1}^\sharp\texttt{(i.0} + 2^{32})\ [\![\texttt{i.0} < \mathsf{intmin}(32)]\!] \tag{9}$$
$$\texttt{bb1}^\sharp\texttt{(i.0)} \to \texttt{bb1}^\sharp\texttt{(i.0} - 2^{32})\ [\![\texttt{i.0} > \mathsf{intmax}(32)]\!] \tag{10}$$
$$\texttt{bb1}^\sharp\texttt{(i.0)} \to \texttt{bb1(i.0)}\ [\![\mathsf{inrange}(\{\texttt{i.0}\})]\!] \tag{11}$$

as an `int`-based TRS. ◊

Arithmetical operations in the constraint: If the constraint φ contains arithmetical operations, then the results of these operations need to be normalized before the constraint can be evaluated. This can be done by adding "dummy" variables for these arithmetic expressions. If φ contains the maximal arithmetic expressions $q_1, \ldots, q_l$, then the rewrite rule ρ is first converted into

$$\begin{aligned}
f(x_1,\ldots,x_n) &\to f^\dagger(x_1,\ldots,x_n,q_1,\ldots,q_l)\ [\![\mathsf{inrange}(\mathcal{V}(\rho))]\!]\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to f^\dagger(x_1,\ldots,x_n,y_1+2^{\|y_1\|},\ldots,y_l)\ [\![y_1 < \mathsf{intmin}(\|y_1\|)]\!]\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to f^\dagger(x_1,\ldots,x_n,y_1-2^{\|y_1\|},\ldots,y_l)\ [\![y_1 > \mathsf{intmax}(\|y_1\|)]\!]\\
&\vdots\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l+2^{\|y_l\|})\ [\![y_l < \mathsf{intmin}(\|y_l\|)]\!]\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l-2^{\|y_l\|})\ [\![y_l > \mathsf{intmax}(\|y_l\|)]\!]\\
f^\dagger(x_1,\ldots,x_n,y_1,\ldots,y_l) &\to g(p_1,\ldots,p_m)\ [\![\widehat{\varphi} \wedge \mathsf{inrange}(\mathcal{V}(\rho)\cup\{y_1,\ldots,y_l\})]\!]
\end{aligned}$$

were $f^\dagger$ is a fresh function symbol and $\widehat{\varphi}$ is obtained from φ by replacing q_i by y_i for all $1 \le i \le l$. Next, the last newly generated rewrite rule is converted as in the previous paragraph.

Optimizations: Notice that the $g^\sharp$-rules (and the $f^\dagger$-rules) essentially use loops for the normalization. Often, a small upper bound on the number of needed loop iterations is known. For instance in $\texttt{i.0} + 1$ from Exa. 8, the correction $2^{\|\texttt{i.0}\|}$ needs to be applied at most once. Thus, the loop for the variable corresponding to $\texttt{i.0} + 1$ can be eliminated by applying the correction zero or one times. Furthermore, $\texttt{i.0} + 1$ can only overflow but never underflow, i.e., an addition of the correction $2^{\|\texttt{i.0}\|}$ does not need to be considered at all.

Example 9. Continuing Exa. 8, the rewrite rules (8)–(11) can be replaced by rewrite rules obtained by combining rewrite rules (8) and (11) and rewrite rules

(8), (10), and (11) into new rewrite rules (these are all possible ways to execute the normalization loop zero or one times for a possible overflow). Then,

$$\begin{aligned}
&\mathtt{entry(i)} \to \mathtt{bb1(i)}\ [\![\mathsf{inrange}(\{\mathtt{i}\})]\!]\\
&\mathtt{bb1(i.0)} \to \mathtt{bb(i.0)}\ [\![\mathtt{i.0} > 0 \wedge \mathsf{inrange}(\{\mathtt{i.0}\})]\!]\\
&\mathtt{bb1(i.0)} \to \mathtt{return()}\ [\![\neg(\mathtt{i.0} > 0) \wedge \mathsf{inrange}(\{\mathtt{i.0}\})]\!]\\
&\mathtt{bb(i.0)} \to \mathtt{bb1(i.0} + 1)\ [\![\mathsf{inrange}(\{\mathtt{i.0}\}) \wedge \mathsf{inrange}(\{\mathtt{i.0} + 1\})]\!]\\
&\mathtt{bb(i.0)} \to \mathtt{bb1(i.0} + 1 - 2^{32})\ [\![\mathsf{inrange}(\{\mathtt{i.0}\}) \wedge \mathtt{i.0} + 1 > \mathsf{intmax}(32)\\
&\qquad\qquad \wedge \mathsf{inrange}(\{\mathtt{i.0} + 1 - 2^{32}\})]\!]
\end{aligned}$$

is obtained. Termination of this `int`-based TRS (or the `int`-based TRS from Exa. 8) is easily established using the techniques from Sect. 3.3, thus establishing termination of the C program using bitvector arithmetic. ◊

5 Experimental Results and Evaluation

In order to assess the practicality of the proposed method, we have implemented it in the termination prover `KITTeL` [13]. For an empirical evaluation, we ran `KITTeL` on the 61 examples presented in [8, Sect. 4.2].[6] These examples were extracted from the *Windows Driver Development Kit*. Of these 61 examples, 51 are terminating and 10 are nonterminating using bitvector arithmetic. Since [8] also provides experimental results for an implementation of their methods, a direct comparison is easily possible.[7] Furthermore, these examples were also used as benchmarks in [26] in order to evaluate an SMT solver.[8] These results are contained in the evaluation as well.

All tools were (assumed to be) run with a timeout of 60s for each example. A summary of the results is given in the following table.[9] In this table, "yes" means a successful termination proof, "maybe" means that execution stopped before the timeout without producing a successful termination proof, and "timeout" means that the timeout was reached. The calculation of average times only takes "yes" and "maybe" answers into account, whereas the average "yes" time only takes "yes" answers into account.

[6] These are all examples considered in [8] for which the source code is available at `http://www.cprover.org/termination/ranking/index.shtml`.

[7] Due to different machines used for the experiments ([8] uses an 8-core Intel® Xeon® 3GHz with 16GB of RAM, whereas our experiments where performed on a 2-core Intel® Core™ 2 Duo 2.4GHz with 4GB of RAM), the comparison of the runtimes is not completely accurate.

[8] A "sat" in [26] means that termination could be shown whereas an "unsat" means that termination could not be shown.

[9] Complete results for `KITTeL` can be found at `http://baldur.iti.kit.edu/~falke/kittel-bitvector/`

	KITTeL	[8], quantifier elimination	[8], SAT solver	[8], QBF solver	[8], SMT solver [26]
# yes	38	30	34	8	27
# maybe	8	24	26	11	14
# timeout	15	7	1	42	20
average time	6.6s	10.8s	2.2s	7.1s	11.8s
average "yes" time	3.3s	11.1s	2.7s	13.4s	12.2s

To summarize the results, KITTeL is more successful than the most successful method from [8] (even in combination with [26]) and needs a comparable average "yes" time. The higher number of timeouts and the higher average time of KITTeL is due to the modular analysis loop used for the termination analysis of int-based TRSs within KITTeL. Whereas the methods from [8] give up as soon as they encounter a path for which they cannot find a ranking function, KITTeL can usually continue its analysis loop by combining rewrite rules (in the evaluation, the number of applications of this technique was restricted to 15).

6 Conclusions

We have shown how the method presented in [13] can be extended to show termination of programs using bitvector arithmetic. In particular, the wrap-around behavior caused by under- and overflows is correctly modeled. Bitwise operations such as and and or, as well as arithmetical operations such as division and remainder, are soundly approximated. This way, unsound and incomplete abstractions caused by identifying bitvectors with integers and bitvector arithmetic with integer arithmetic (as is done by nearly all current methods for proving termination of imperative programs) are avoided and the behavior of an imperative program is modeled in the way it is executed on the computer.

An implementation of the proposed method in the tool KITTeL [13] has been evaluated on 61 examples extracted from the *Windows Driver Development Kit* that were also used in [8]. The performance of KITTeL on these examples is very encouraging—better than the performance of the tool presented in [8]. In future work, we plan to extend KITTeL in order to reason about termination due to the traversal of well-formed linked data structures on the heap. Furthermore, we plan to explore refined sound approximations for arithmetic and bitwise operations. Finally, since the modeling of bitvector arithmetic adds overhead in comparison to integer arithmetic, the development of a CEGAR-like approach which starts using integer arithmetic and gradually refines the model to use bitvector arithmetic is a promising direction for future research.

Acknowledgments. We thank the anonymous reviewers for helpful comments and for pointing us to the evaluation contained in [26].

References

1. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 2–18. Springer, Heidelberg (2008)
2. Babić, D., Musuvathi, M.: Modular arithmetic decision procedure. Tech. Rep. TR-2005-114, Microsoft Research Redmond (2005)
3. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear Ranking with Reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination Analysis of Integer Linear Loops. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 488–502. Springer, Heidelberg (2005)
5. Brockschmidt, M., Otto, C., Giesl, J.: Modular termination proofs of recursive Java bytecode programs by term rewriting. In: RTA 2011(2011)
6. Colón, M., Sipma, H.: Synthesis of Linear Ranking Functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001)
7. Colón, M., Sipma, H.: Practical Methods for Proving Program Termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
8. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking Function Synthesis for Bit-Vector Relations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 236–250. Springer, Heidelberg (2010)
9. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction Refinement for Termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
10. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI 2006, pp. 415–426 (2006)
11. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
12. Falke, S., Kapur, D.: A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 277–293. Springer, Heidelberg (2009)
13. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: RTA 2011, pp. 41–50 (2011)
14. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination Analysis with Compositional Transition Invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
15. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004, pp. 75–88 (2004)
16. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. ACM TOPLAS 29(5), 29:1–29:27 (2007)
18. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java bytecode by term rewriting. In: RTA 2010, pp. 259–276 (2010)
19. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)

20. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS 2004, pp. 32–41 (2004)
21. Simon, A., King, A.: Taming the Wrapping of Integer Arithmetic. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 121–136. Springer, Heidelberg (2007)
22. Sinz, C., Falke, S., Merz, F.: A precise memory model for low-level bounded model checking. In: SSV 2010 (2010)
23. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for Java bytecode based on path-length. ACM TOPLAS 32(3), 8:1–8:70 (2010)
24. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop Summarization and Termination Analysis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 81–95. Springer, Heidelberg (2011)
25. Wang, B.Y.: On the Satisfiability of Modular Arithmetic Formulae. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 186–199. Springer, Heidelberg (2006)
26. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: Efficiently solving quantified bit-vector formulas. In: FMCAD 2010, pp. 239–246 (2010)