# Non-Termination Checking for Imperative Programs

Helga Velroyen[1] and Philipp Rümmer[2]

[1] Department of Computer Science
RWTH Aachen University of Technology
helga.velroyen@rwth-aachen.de
[2] Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University
philipp@chalmers.se

**Abstract.** While termination checking tailored to real-world library code or frameworks has received ever-increasing attention during the last years, the complementary question of *disproving* termination properties as a means of debugging has largely been ignored so far. We present an approach to automatic non-termination checking that relates to termination checking in the same way as symbolic testing does to program verification. Our method is based on the automated generation of invariants that show that terminating states of a program are unreachable from certain initial states. Such initial states are identified using constraint-solving techniques. The method is fully implemented on top of a program verification system and available for download. We give an empirical evaluation of the approach using a collection of non-terminating example programs.

## 1 Introduction

Termination properties of programs are crucial for liveness and safety: a piece of software which does not terminate can have vast consequences, especially when employed in critical environments or wide-spread. The latter concerns in particular library code or frameworks, whose specific use is often unknown at the time of development. Non-termination bugs can be very subtle and hide long before they take effect in productivity situations.

Although the concept of formally proving termination properties has been known and investigated for a long time, the last years have seen intensified research on how to check the termination of real-world code [1, 2]. During the same time, however, the complementary field of showing the potential *non-termination* of programs as a means of debugging has largely been ignored. This is a surprising situation, because programs under development are prone to contain defects. In this context, direct attempts to find those bugs might be more successful and more useful than to learn from failed correctness or termination proofs.

Traditional dynamic techniques of testing program behavior by means of concrete execution are not adequate to show non-termination (they can nevertheless provide valuable hints). As a consequence, although the purpose of

non-termination analysis is more related to testing than to program verification, in most cases the usage of symbolic reasoning cannot be avoided. In the present paper, we introduce an approach to automatic non-termination checking that relates to termination checking in the same way as symbolic testing does to program verification. The method has been implemented on top of a general-purpose program verification system. Experiments using a database of non-terminating programs indicate that it can be a useful tool for detecting termination defects early during software development.

Showing the non-termination of a program consists of two parts: (i) to prove that a potential loop in a program is reachable from some initial state, and (ii) to prove that the potential loop can indeed cause non-termination. We use constraint solving techniques to achieve the first part, following the approach described in [3]. For the second part, we introduce an algorithm to synthesise invariants that show that the found loop is never exited and that terminating states of the program are therefore unreachable. Our approach is based on two main techniques, a template method for generating invariants (together with constraint solving) and refinement (strengthening) of invariants based on counterexamples. Because our experiments show that invariants for proving non-termination are typically much smaller than invariants for proving partial correctness, we believe that this yields a practical procedure for constructing non-termination proofs.

*The paper is organised as follows:* In Sect. 2 we define the programming language that is analysed in the whole paper. Sect. 3 introduces the logic and the calculus that we use to reason about programs, which is the basis for an effective algorithm in Sect. 4. An empirical evaluation of our approach is given in Sect. 5. Finally, we list related work in Sect. 6 and conclude in Sect. 7.

## 2 Preliminaries

We assume that the reader is familiar with classical first-order logic and Gentzen-style sequent calculi, see [4] for an introduction. For sake of simplicity, all considerations of this paper are done in the context of a simple while-language that operates on the (infinite) domain of integers. The generalisation to other imperative languages is mostly straightforward, and, in our experience, occurring problems tend to be orthogonal to the task of proving non-termination. More details are given in [5, 3].

In order to introduce the while-language, we first assume a fixed vocabulary $\Sigma$ of functions and predicates (with fixed arity) that describe the native side-effect-free operations that are available, as well as a fixed set $V_p$ of program variables. The set $\Sigma$ is supposed to contain at least literals and the standard operations on integers $(0, 1, -1, \ldots, +, -, \cdot, =, <, \leq)$. Ground terms, ground formulae and programs are then inductively defined by the following grammars:

$$
\begin{aligned}
t_\mathrm{g} &::= v \mid f(t_\mathrm{g}, \ldots, t_\mathrm{g}) \\
\phi_\mathrm{g} &::= \mathit{true} \mid \mathit{false} \mid \phi_\mathrm{g} \wedge \phi_\mathrm{g} \mid \neg\phi_\mathrm{g} \mid \cdots \mid p(t_\mathrm{g}, \ldots, t_\mathrm{g}) \\
\alpha &::= \alpha \,;\, \ldots \,;\, \alpha \mid v = t_\mathrm{g} \mid \texttt{if}\ (\phi_\mathrm{g})\ \alpha\ \texttt{else}\ \alpha \mid \texttt{while}\ (\phi_\mathrm{g})\ \alpha
\end{aligned}
$$

where $f \in \Sigma$ ranges over functions, $p \in \Sigma$ over predicates and $v \in V_p$ over program variables.

*Semantics of Programs.* Because only the integers are considered as domain, a *structure* is a pair $S = (\mathbb{Z}, I)$ consisting of the set $\mathbb{Z}$ of integers and an *interpretation* $I$ with $I(f) : \mathbb{Z}^n \to \mathbb{Z}$ if $f \in \Sigma$ is a function of arity $n$ and $I(p) \subseteq \mathbb{Z}^n$ if $p \in \Sigma$ is a predicate of arity $n$. Only those structures are considered in which the standard integer operations from above (like $0, 1, -1, +, \ldots$) have their usual meaning. A *program variable assignment* is a mapping $\gamma : V_p \to \mathbb{Z}$. The space of all program variable assignments is denoted by $PA = V_p \to \mathbb{Z}$. While-programs $\alpha$ are evaluated in structures $S$ and denote partial mappings $[\![\alpha]\!]^S : PA \rightharpoonup PA$ from program variable assignments to program variable assignments:

$$[\![\alpha]\!]^S(\gamma) \;=\; \begin{cases} \gamma' & \alpha \text{ terminates in state } \gamma' \text{ when started in } \gamma \\ \bot & \alpha \text{ does not terminate when started in } \gamma \end{cases}$$

Given an evaluation function $val_{S,\gamma}$ for ground terms and formulae, which is defined as is common for first-order logic (cf. [4]), the concrete definition of $[\![\alpha]\!]^S$ follows the lines of denotational semantics (for instance, [6]).

## 3 Proving Non-Termination: The Calculus Level

We introduce our approach to non-termination detection in two parts: in this section, we describe the logic and the calculus to reason about programs. Based on this declarative framework, Sect. 4 defines an algorithm (a proof procedure) for automatically detecting non-termination.

**Dynamic Logic for the While-Language (WhileDL).** First-order dynamic logic (DL) [7] is a multi-modal extension of first-order predicate logic, in which modal operators are labelled with programs. Most importantly, given a program $\alpha$ and a formula $\phi$, a *box-formula* $[\alpha]\,\phi$ expresses that $\phi$ holds in each final state of $\alpha$. This paper uses a version of dynamic logic for the simple while-language [7] that is enriched with an explicit operator for simultaneous substitutions called *updates* [8, Sect. 3]. Updates allow us to present some of the techniques of this papers in a simpler way, but also simplify the generalisation to more involved languages like Java [5, 3, 8].

We assume the same vocabulary $\Sigma$ and the same set $V_p$ of program variables as in Sect. 2, but in addition we define a disjoint set $V_l$ of *logical variables* that can occur in formulae and terms (outside of programs). Because some of our rules need to introduce fresh function symbols, we assume that $\Sigma$ contains infinitely many functions for each arity $n$. Extending the grammar from Sect. 2, arbitrary terms, formulae and updates are then defined by:

$$\begin{aligned} t \;&::=\; t_{\mathrm{g}} \mid x \mid f(t, \ldots, t) \mid \{\, U \,\}\, t \\ \phi \;&::=\; \phi_{\mathrm{g}} \mid \phi \wedge \phi \mid \neg\phi \mid \;\cdots\; \mid p(t, \ldots, t) \mid [\alpha]\,\phi \mid \{\, U \,\}\, \phi \\ U \;&::=\; v := t \mid U, \ldots, U \end{aligned}$$

where $f \in \Sigma$ ranges over functions, $p \in \Sigma$ over predicates, $x \in V_l$ over logical variables and $v \in V_p$ over program variables.

In order to define the semantics of terms, formulae and updates, besides structures $S = (\mathbb{Z}, I)$ and program variable assignments $\gamma \in PA$ we also need *logical variable assignments* $\beta : V_l \to \mathbb{Z}$. The denotation $[\![U]\!]^{S,\beta} : PA \to PA$ of an update $U$ is a total operation on program variable assignments:

$$[\![v_1 := t_1, \ldots, v_k := t_k]\!]^{S,\beta}(\gamma)(w) \quad = \quad \begin{cases} val_{S,\beta,\gamma}(t_i) & w = v_i \text{ and} \\ & \quad w \notin \{v_{i+1}, \ldots, v_k\} \\ \gamma(w) & w \notin \{v_1, \ldots, v_k\} \end{cases}$$

This means that the assignments of an update are executed in parallel, and that assignments that syntactically occur later can override the effects of earlier assignments ($v_j := t_j$ will override $v_i := t_i$ for $v_i = v_j$ and $j > i$).

The evaluation $val_{S,\beta,\gamma}$ of terms and formulae is mostly defined as it is common for first-order predicate logic. Formulae are mapped into a Boolean domain, where tt stands for semantic truth. The cases for programs and updates are:

$$\begin{aligned} val_{S,\beta,\gamma}([\alpha]\,\phi) \quad &= \quad \begin{cases} val_{S,\beta,[\![\alpha]\!]^S(\gamma)}(\phi) & \text{if } [\![\alpha]\!]^S(\gamma) \text{ is defined} \\ \text{tt} & \text{otherwise} \end{cases} \\ val_{S,\beta,\gamma}(\{\,U\,\}\,\phi) \quad &= \quad val_{S,\beta,[\![U]\!]^{S,\beta}(\gamma)}(\phi) \end{aligned}$$

We interpret free logical variables $x \in V_l$ existentially: a formula $\phi$ is *valid* iff for each structure $S$ and each program variable assignment $\gamma \in PA$ there is a variable assignment $\beta : V_l \to D$ such that $val_{S,\beta,\gamma}(\phi) = \text{tt}$. Likewise, a sequent $\Gamma \vdash \Delta$ is called valid iff $\bigwedge \Gamma \to \bigvee \Delta$ is valid. Free variables are used to express symbolic program inputs and as parameters in loop invariants and serve as an interface to constraint solving (see below for more details).

**Characterisation of Non-Termination.** Because box-formulae $[\alpha]\,\phi$ are trivially rendered true by a diverging program $\alpha$, we can express non-termination by asserting *false* as post-condition: $[\alpha]\,\mathit{false}$. This means that, given a structure $S$, $val_{S,\gamma}([\alpha]\,\mathit{false}) = \text{tt}$ holds for exactly those initial states $\gamma \in PA$ for which $\alpha$ diverges.

In order to express non-termination for some *arbitrary* initial state, it is necessary to quantify the variables occurring in $\alpha$ existentially, following the approach from [3]. For the while-language, this is done by prefixing the formula from above with an update that assigns arbitrary values to all program variables in $\alpha$:

$$\{\, v_1 := x_1, \ldots, v_n := x_n \,\}\,[\alpha]\,\mathit{false} \tag{1}$$

where $v_1, \ldots, v_n \in V_p$ are the variables occurring in $\alpha$ and $x_1, \ldots, x_n \in V_l$ are fresh logical variables. (1) is valid iff there are initial states from which $\alpha$ diverges.

**A Sequent Calculus for WhileDL.** To reason formally about the non-termination of programs, we introduce a Gentzen-style sequent calculus for WhileDL that follows closely the calculi in [3, 8]. Fig. 1 contains the most important calculus rules, which can be categorised as program-independent *first-order rules* (the upper part of the figure) and *symbolic execution rules*.

The rule Assign turns assignments into updates, which subsequently can be merged with the former preceding update $U$ and simplified. The simplification and application of updates is performed by the rewriting rules in Fig. 2, which propagate updates in formulae or terms downwards until they can be applied to program variables like substitutions.

In If, a case analysis for an if-statement is performed by splitting on the branch predicate $\psi$ evaluated in the current program state $U$. The invariant rule While is a simplified version of the rule for Java defined in [8, Chap. 3]. In While, the erasure of side formulae is avoided with the help of *anonymising updates* $A_1$, $A_2$ that assign unspecified values to all variables that can be modified by the loop body $\alpha$. More formally, given that (i) $v_1, \ldots, v_n \in V_p$ are the variables that occur as left-hand sides of assignments in $\alpha$, that (ii) $x_1, \ldots, x_m \in V_l$ are the logical variables that occur in $U$, $\phi$, or $Inv$, and that (iii) $f_1, \ldots, f_n$ are fresh function symbols, we say that the update

$$v_1 := f_1(x_1, \ldots, x_m),\ \ldots,\ v_n := f_n(x_1, \ldots, x_m)$$

is a fresh anonymising update for $\alpha$ with respect to $U, \phi, Inv$. Note, that we need to inject the logical variables $x_1, \ldots, x_m$ as arguments of the functions $f_1, \ldots, f_n$ for exactly the same reasons as in the standard Skolemisation rule (cf. [4]).

Finally, *theory rules* are necessary to handle equality, integers, etc. in the calculus, we refer the reader to [9] for more details. An example proof using the WhileDL calculus is shown below.

When inspecting the calculus rules, it can be observed that all rules but While are local equivalence transformations: for all structures, program variable assignments and logical variable assignments, the conclusion of a rule holds iff all premisses hold. This property is important for us, because it implies that countermodels of an open goal are also countermodels of the initial conjecture (unless While has been applied). In Sect. 4, we use counterexamples that were extracted from open proof goals to refine invariant candidates.

**Incremental Closure of Proofs.** In order to close a proof tree that contains free logical variables, we have to show that the variables can be given values (depending on the considered structure) such that all remaining goals are turned into obviously valid sequents. We apply the idea of *incremental closure* [4, 10] together with the arithmetic constraint language from [3, Sect. 4] to check the existence of such values. The rules in Fig. 3 are responsible for introducing closure constraints for proof goals. If it is possible, in this way, to find compatible closure constraints for *all* proof goals (i.e., the conjunction of the constraints is valid), then it is sound to close the proof.

$$\frac{*}{\Gamma \vdash \mathit{true}, \Delta}\ \text{TRUE-RIGHT} \qquad \frac{\Gamma \vdash \Delta}{\Gamma, \mathit{true} \vdash \Delta}\ \text{TRUE-LEFT}$$

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}\ \wedge\text{-RIGHT} \qquad \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \wedge\text{-LEFT}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \neg\text{-RIGHT} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \neg\text{-LEFT}$$

$$\ldots$$

$$\frac{\Gamma \vdash \{U\}\,\phi, \Delta}{\Gamma \vdash \{U\}\,[\,]\,\phi, \Delta}\ \textsc{Skip} \qquad \frac{\Gamma \vdash \{U\}\,\{v := t\}\,[\ldots]\,\phi, \Delta}{\Gamma \vdash \{U\}\,[v = t\ ;\ \ldots]\,\phi, \Delta}\ \textsc{Assign}$$

$$\frac{\Gamma \vdash \{U\}\,(\psi \to [\alpha_1\ ;\ \ldots]\,\phi), \Delta \qquad \Gamma \vdash \{U\}\,(\neg\psi \to [\alpha_2\ ;\ \ldots]\,\phi), \Delta}{\Gamma \vdash \{U\}\,[\texttt{if}\ (\psi)\ \alpha_1\ \texttt{else}\ \alpha_2\ ;\ \ldots]\,\phi, \Delta}\ \textsc{If}$$

$$\frac{\begin{array}{l}\Gamma \vdash \{U\}\,\mathit{Inv}, \Delta \\ \Gamma \vdash \{U\}\,\{A_1\}\,(\mathit{Inv} \wedge \psi \to [\alpha]\,\mathit{Inv}), \Delta \\ \Gamma \vdash \{U\}\,\{A_2\}\,(\mathit{Inv} \wedge \neg\psi \to [\ldots]\,\phi), \Delta\end{array}}{\Gamma \vdash \{U\}\,[\texttt{while}\ (\psi)\ \alpha\ ;\ \ldots]\,\phi, \Delta}\ \textsc{While}$$

($A_1, A_2$ are fresh anonymising updates for $\alpha$ w.r.t. $U, \phi, \mathit{Inv}$)

**Fig. 1.** Sequent calculus for WhileDL. In the last four rules, the update $\{U\}$ can also be empty and disappear.

$$\begin{array}{rcll}
\{v_1 := t_1, \ldots, v_k := t_k\}\, v_i & \to & t_i & \text{if } v_i \notin \{v_{i+1}, \ldots, v_k\} \\
\{v_1 := t_1, \ldots, v_k := t_k\}\, t & \to & t & \text{if } v_1, \ldots, v_k \text{ do not occur in } t \\
\{U\}\, f(t_1, \ldots, t_n) & \to & f(\{U\}\, t_1, \ldots, \{U\}\, t_n) & \\
\{U\}\, p(t_1, \ldots, t_n) & \to & p(\{U\}\, t_1, \ldots, \{U\}\, t_n) & \\
\{U\}\, \neg\phi & \to & \neg\{U\}\, \phi & \\
\{U\}\, (\phi \wedge \psi) & \to & \{U\}\, \phi \wedge \{U\}\, \psi & \\
\{U\}\, \{v_1 := t_1, \ldots, v_k := t_k\}\, \phi & \to & \{U,\ v_1 := \{U\}\, t_1,\ \ldots,\ v_k := \{U\}\, t_k\}\, \phi & \\
\{\ldots, v := s, \ldots, v := t, \ldots\}\, \phi & \to & \{\ldots, v := t, \ldots\}\, \phi &
\end{array}$$

**Fig. 2.** The main application rules for updates in WhileDL. Further rules to simplify updates can be formulated (cf. [8, Chap. 3]), but are not shown here.

$$\frac{[\,s = t\,]}{\Gamma \vdash s = t, \Delta}\ =\text{-RIGHT} \qquad \frac{[\,s \leq t\,]}{\Gamma \vdash s \leq t, \Delta}\ \leq\text{-RIGHT} \qquad \frac{[\,s \geq t\,]}{\Gamma \vdash s \geq t, \Delta}\ \geq\text{-RIGHT}$$

$$\frac{[\,s \neq t\,]}{\Gamma, s = t \vdash \Delta}\ =\text{-LEFT} \qquad \frac{[\,s > t\,]}{\Gamma, s \leq t \vdash \Delta}\ \leq\text{-LEFT} \qquad \frac{[\,s < t\,]}{\Gamma, s \geq t \vdash \Delta}\ \geq\text{-LEFT}$$

**Fig. 3.** Closure rules for the WhileDL sequent calculus

**Example.** We illustrate the usage of the sequent calculus by proving the non-termination of the following program:

$$\textsc{Lcm} \quad = \quad \begin{cases} a = a_0 \ ; \ b = b_0 \ ; \\ \texttt{while} \ (a \neq b) \ \{ \\ \qquad \texttt{if} \ (a > b) \ b = b + b_0 \ \texttt{else} \ a = a + a_0 \\ \} \end{cases}$$

In case of termination, the post-value of $a$ and $b$ is the least common multiple of the two integers $a_0$, $b_0$. The program fails, however, to handle negative inputs correctly: if the signs of $a_0$ and $b_0$ are different, for instance, the program does not terminate. To prove this formally, we instantiate (1) with Lcm and construct a proof tree (Fig. 4).

The only step in the course of the proof that requires creativity is the choice of the formula *Inv* that is used as invariant when applying the rule While (our technique for synthesising such formulae is described in the next section). In terms of the program execution, *Inv* has to describe a set of program states that (i) is entered when Lcm reaches the while-loop, (ii) is not left during the execution of the loop, and (iii) does not contain any states in which the loop guard becomes false. We chose $a < b$ as invariant in this example, but similar proofs can be given for the invariants $a < 0 \wedge b > 0$ or $a \neq b$. In all cases, the technique of incremental closure has to be used to determine some initial state (i.e., values of the variables $a_0$, $b_0$) for which the chosen formula *Inv* actually *is* an invariant and the proof can be closed. The closing constraint in Fig. 4 is $[\, x_a < 1 \wedge x_a < x_b \,]$, which means that we have proven the non-termination for initial states $(a_0, b_0)$ like $(0, 1), (0, 2), (-10, -5)$, etc.

## 4 Automatically Detecting Non-Termination

In our work, we developed an algorithm to identify non-terminating programs automatically. It has two components, an invariant generator and a theorem prover. The theorem prover is used to prove formulae which state the non-termination of a program. This done by construction of proof trees using the calculus rules and incremental closure, described in Sect. 3. The other component, the invariant generator, is used to provide and refine invariants for the theorem prover. It was used to construct the invariant $a < b$ from the previous section in a systematic way.

The idea of the algorithm is to construct a non-termination proof as described in the preceding section. The essential part of a non-termination proof is the invariant which is used in the application of the While rule. Our algorithm tries to find this invariant by repeatedly constructing proof attempts. In each iteration a different invariant is used, starting with the formula *true*, representing that the prover has no knowledge about the invariant at start up. After each failed proof attempt, the incomplete proof tree is examined. The retrieved information from this examination is then used to refine the invariant. There are several ways of refinement of which one uses template variables for the invariants.

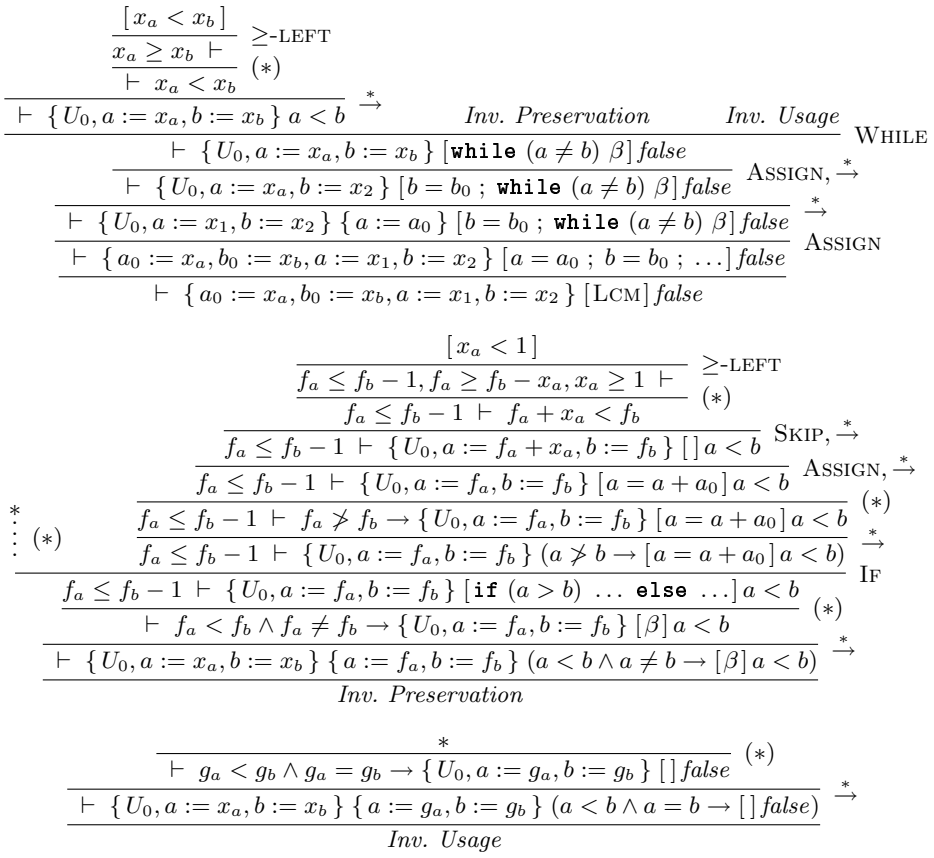


**Fig. 4.** Proof for the (potential) non-termination of the program LCM using the invariant $a < b$. The proof can be closed with the constraint $[\, x_a < 1 \wedge x_a < x_b \,]$, which describes a set of initial states that causes LCM to diverge. We write $\beta$ for the body of the while-loop, $f_a$, $f_b$, $g_a$, $g_b$ as abbreviation for the Skolem terms $f_a(x_a, x_b)$, $f_b(x_a, x_b)$, $g_a(x_a, x_b)$, $g_b(x_a, x_b)$, and $U_0$ as abbreviation for the update $a_0 := x_a, b_0 := x_b$. Rewriting steps to apply updates are denoted by $\overset{*}{\rightarrow}$, whereas $(*)$ means that rules for propositional and arithmetic reasoning are applied which are not shown in detail.

| It. | cur. Inv. | Open goals | Queue after step 5 of algorithm |
|---|---|---|---|
| 1 | *true* | $a = b \vdash$ | $b > a$, $b < a$, $b < U_b$, $a < U_a$, $b > L_b$, $a > L_a$, $a \neq b$ |
| 2 | $b > a$ | none | $b < a$, $b < U_b$, ... |

**Fig. 5.** Application of the algorithm on LCM. Technically, $a$ and $b$ in the open goals are Skolem terms like $f_a(x_a, x_b)$ in Fig. 4, which have to be translated back to obtain invariants in terms of the program variables. In iteration 2, the non-termination proof can be closed with the constraint $[\, x_a < x_b \wedge x_a < 1 \,]$. The result expresses that LCM does not terminate if the initial value of $a_0$ is less than that of $b_0$ and not positive.

A positive result of the algorithm is a successful non-termination proof of the program together with a description of a set of input values for which the loop of the program runs forever.

*Note on Nested Loops.* The algorithm as it is described here is only applicable to single, unnested loops. As it is always possible to transform nested loops into unnested ones, this is no real restriction. Besides, in [5] we describe how our algorithm can be adapted so that it directly works on nested loops.

**Outline of the Algorithm.** Let $\alpha$ be the program whose termination is in question. The input of the algorithm is $\alpha$'s source code, which is inserted into a WhileDL formula $\phi$ (formula (1) in Sect. 3) which states that there are inputs for which $\alpha$ does not terminate.

*Initialisation*

1. The formula $\phi$ is handed over to the theorem prover. The proof procedure is invoked and constructs a proof tree in which the program is symbolically executed until the execution reaches the loop.

*Iteration*

2. The proof procedure applies the invariant rule WHILE (Fig. 1). The invariant $Inv_{cur}$ which is used in the invariant rule's application is chosen from a queue of invariants. Initially there is only $Inv_{cur} \equiv true$ in the queue.
3. The proof procedure keeps on constructing the proof as far as possible without human interaction.
4. If the proof procedure can close the proof, the algorithm terminates with the result that the program does not terminate. If the proof cannot be closed, the open goals of the proof are extracted and handed over to the invariant generator.
5. The invariant generator inspects the formulae of the open goals. The obtained information is used to refine the invariant candidate to create one or more new candidates, which are then added to the queue.

The algorithm repeats step 2 to 5 iteratively, each time using one of the invariant candidates from the invariant queue. The iterations are carried out until one of these events occurs: the proof can be closed with the help of the invariant candidate, the algorithm runs out of new invariant candidates or a maximum number of iterations is reached. In case of a successful termination of the algorithm, it outputs the invariant used for the final proof, together with the (consistent) closing constraint.

There are three parts of step 5 of the algorithm that we like to describe in more detail. The first is the actual creation of the invariants.

**Invariant Creation.** There are different methods to create new invariants from the open goals of failed proofs. Assume that we obtained the open goal

$$\phi_1, \ldots, \phi_n \vdash \psi_1, \ldots \psi_m$$

where $\phi_i$ and $\psi_i$ are WhileDL-formulae. Given such an open goal, the invariant generator creates invariant fragments $\rho$ which are conjunctively added to the invariant $Inv_{cur}$ which was used in the current iteration to obtain a new invariant $Inv_{new} = Inv_{cur} \wedge \rho$. The invariant fragments are created by the following operations:

- ADD. A formula $\psi_i$ in the succedent states a situation in which there is a problem with the non-termination proof when $\psi_i$ does not hold. Most often that means that in this situation the loop actually terminates. We exclude this situation by setting $\rho = \psi_i$.
- NEGADD. A formula $\phi_i$ in the antecedent means that there is a problem with the non-termination in the situation where $\phi_i$ holds. Here, the same idea applies as for formulae in the succedent, but in this case we have to negate it before we add it to the old invariant, which means $\rho = \neg\phi_i$.
- INEQ. In case a formula $\phi_i$ of the antecedent is of form $\phi_i \equiv a = b$, we do not only add the negation as in NEGANDADD, but an inequality. That means from $a = b$ we obtain two fragments $\rho_1 \equiv a \geq b$ and $\rho_2 \equiv a \leq b$, yielding two different new invariants.
- INEQVAR. Often it is useful to express that *there are* upper or lower bounds for an expression rather than specifically setting one like in INEQ. This is done through the introduction of free logical variables. Those variables stand for particular but not yet specified values. For each term in the open goal, we provide two new variables $U$ and $L$, one for the upper and one for the lower bound. Thus, for each term $t_k$ occurring in one $\phi_i$, we obtain two fragments $\rho_k^u \equiv t_k \leq U_k$ and $\rho_k^l \equiv t_k \geq L_k$. The values for the new variables are estimated by the constraint solver of the proof procedure.

The latter two creation methods are of course only applicable if $a$, $b$ and $t_k$ are expressions of an ordered type, in our case integers.

**Invariant Filtering.** In the process of invariant creation, sometimes invariant candidates are created that are not helpful in the search of a non-termination invariant. This is due to the fact that these methods are applied "blindly" without actually examining the old invariant candidate. Therefore, after the creation of invariants in step 5 of the algorithm, we filter out those candidates which are obviously useless:

- *Inconsistent Invariant.* A newly created invariant candidate can be equivalent to the formula *false*. Because the first property of non-termination invariants is that the invariant must hold before the loop execution, it is dismissed.

- *Equivalence to Previous Invariants.* A new invariant candidate can be equivalent to a candidate that was already created and/or used in an earlier iteration. Dismissal of these candidates avoid unnecessary calculations and thus save resources.
- *Impossible Closure of the Init-branch.* The application of the invariant rule makes the proof branch into three branches. The first branch proves that the invariant holds when the loop is reached in the execution of the program. In the refinement process, invariant candidates might be created that do not hold in the beginning of the loop, even if they are satisfiable in general. Once we have created an invariant candidate which prevents the first branch from closing, it does not make sense to refine any further: refinement would only strengthen the candidate even more.[3]
- *Complexity.* For performance reasons, we set a limit on the complexity of formulae to keep the runtime at a reasonable level.

**Invariant Scoring.** In each iteration of the algorithm, when the invariant candidates are created and filtered in step 5 still a lot of invariants can remain. In order to traverse the search space of invariants in a reasonable way, we have to queue invariants according to their probable usefulness for non-termination proofs.

We estimate this usefulness by several criteria and express it in a score, which is a real number between 0 and 1. The lower the score is, the more the invariant is preferred in the queue. The score is calculated as a weighted average of scores for each of the following criteria.

- *Complexity.* In order to find the most general description of a set of critical inputs, we prefer simple invariants to complex ones. The complexity is measured in both the term depth and the number of operators of the invariant.
- *Existence of Free Variables.* The creation method IneqVar is a strong tool (and sometimes the only effective one) to find the desired invariant. The problem with free variables is that in cases where they do *not* lead to a closed proof, they tend to lead to even bigger open proofs. It is reasonable to prefer invariant candidates that do not contain free variables to those who do in order to keep the number of newly created candidates as low as possible.
- *Multiple Occurrence of Formulae.* In an open proof, sometimes the same formulae occur in several open goals. We prefer invariant candidates made from those formulae to others, because if the candidate makes the algorithm close branches, it will close several branches in the same proof.
- *Reoccurring Formulae.* Formulae which occurred in open proofs in several iterations of the algorithm might be suitable candidates for the next invariant, because they hint to situations where the non-termination proof repeatedly failed.

[3] The filtering of inconsistent invariants is subsumed in this filter. We kept it in the list of filters, because checking for inconsistency is easier than for closure of the initial branch. So, for performance reasons it is useful to first check only for consistency before examining the closability of the first branch.

– *Proof Size.* We presume that the smaller an open proof is (measured in the number of open goals) the closer it is to being closed. Therefore we prefer formulae which come from small open proofs to those from big open proofs.

Experiments have shown that the choice and weighting of the criteria is extremely important for the search in the space of invariants. In our work, we ran several experiments to test the impact of different heuristics, the results of which are given in Sect. 5 and [5].

**Examples.** We apply our algorithm to the example programs FIB and LCM, of which the latter one was introduced in Sect. 3 already. For the sake of simplicity, we assume that for scoring of the invariants only the criteria of complexity is applied.

*Example* LCM. Fig. 5 shows how the algorithm works on LCM. In this case all presented creation methods are used.

*Example* FIB. Given a Fibonacci number $n$ as input, FIB calculates how many calculation steps are necessary in the series of Fibonacci numbers to reach $n$. The result is stored in variable $c$. In case $n$ is not a Fibonacci number, the program does not terminate.

$$\text{FIB} \quad = \quad \begin{cases} i = 0 \; ; \; j = 1 \; ; \; t = 0 \; ; \; c = 2 \; ; \\ \texttt{while} \; (j \neq n) \; \{ \\ \qquad t = j + i \; ; \; i = j \; ; \; j = t \; ; \; c = c + 1 \\ \} \end{cases}$$

In contrast to LCM, the algorithm needs several refinement steps (Fig. 6) to prove the non-termination of FIB. The input variable $n$ is associated with the free logical variable $x_n$. This time, we used only the creation methods ADD, NEGADD, INEQ, together with the complexity scoring criterion. We abstained from showing the creation method INEQVAR, because it increases the number of necessary iterations too much to be shown here. However, we did run the same experiment with INEQVAR and will present the results in the following section.

**Properties of the algorithm** We would like to have a closer look at the properties of the algorithm which we presented here.

– *Soundness.* The algorithm is sound for non-termination: it will never identify a terminating program as non-terminating. This is an immediate consequence of the soundness of the calculus from Sect. 3, because non-termination is only reported if it was possible to construct a proof for it. Applied to a terminating program, the algorithm will fail to find such a proof and will output that it was not able to prove non-termination.

– *Incompleteness.* Unfortunately, but expectedly, both our calculus and the algorithm are not complete for non-termination: there are programs that do not terminate for some inputs, but there is no proof of this fact in the calculus from Sect. 3. This is implied by the soundness, because the set of programs that do not terminate for some inputs is not recursively enumerable.[4] Because the algorithm is based on heuristics, it might also fail to find existing non-termination proofs for a program, of course.
– *Automation.* The algorithm works fully automatic, in the sense that no manual "human" actions are necessary to obtain the results.
– *Determinism.* The algorithm is deterministic, because for the same input it always produces the same results. The indeterministic calculus which forms the base of the prover is made deterministic by choice of heuristics and prioritisation.
– *Termination.* Our algorithm itself always terminates. This is ensured by setting an upper limit for the number of iterations, and by limiting the size of proofs in the calculus from Sect. 3 that are constructed. Of course these limits have to be chosen carefully, because the lower they are the fewer non-terminating programs can be identified.

## 5 Experiments

We implemented the algorithm, which we presented in Sect. 4 in particular we wrote the part of the invariant generator and used the software KeY [8] as theorem prover. Both are written in Java. Since there was no publicly available standardised example set of non-terminating programs, we built up one to estimate the quality of our approach and test different heuristics.

*Example Set.* Our example set consists of 55 programs, of which 53 are known to be non-terminating for all or some input values, one whose termination behavior is not fully known and one which is terminating. All programs are written in a fragment in Java, which captures the functionality of the While language which we described in Sect. 2. They have between one and five variables and up to 25 lines of code. We chose them either because they represent typical programming errors or because they reveal very tricky non-termination behavior.

*Results of the Experiments.* We tested different settings concerning creation and scoring of invariants in several experiments [5]. Our software could solve 41 of the 55 examples automatically, but not more than 37 with one setting. This fact shows how sensitive the algorithm's heuristics are.

Some of the experiments were used to estimate the usefulness of the different creation methods of Sect. 4, in particular the method IneqVar. Experiments who included free logical variables as invariant templates could solve about 20% more problems than those who did not. Free variables are obviously a strong tool

[4] Note that the set of programs that terminate for *all* possible inputs is not recursively enumerable either.

| It. | cur. Inv. | Open goals |
|---|---|---|
| 1 | $Inv_1 \equiv true$ | $j = x \vdash$ |
| 2 | $Inv_2 \equiv j > x$ | $x \geq 1 \vdash$ , $j \leq x - i, i \leq -1, j \geq 1 + x \vdash$ |
| 3 | $Inv_3 \equiv j < x$ | $x \leq 1 \vdash$ , $i \geq 1, j \geq x - i, j \leq -x \vdash$ |
| 4 | $Inv_4 \equiv j \neq x$ | $x = 1 \vdash$ , $j = x - i, i = 0 \vdash$ |
| 5 | $Inv_5 \equiv j > x \wedge x < 1$ | $x \geq 1 \vdash$ , $j \leq x - i, x \leq 0, i \leq -1, j \geq 1 + x \vdash$ |
| 6 | $Inv_6 \equiv j > x \wedge x > -1$ | none |

The next invariants to be tried:

| | |
|---|---|
| $Inv_7 \equiv j < x \wedge x > 1$ | $Inv_{14} \equiv j \neq x \wedge j > x - i$ |
| $Inv_8 \equiv j < x \wedge i < 1$ | $Inv_{15} \equiv j > x \wedge x < 1 \wedge x > 0$ |
| $Inv_9 \equiv j \neq x \wedge i = 0$ | $Inv_{16} \equiv j > x \wedge j > x - i$ |
| $Inv_{10} \equiv j \neq x \wedge x > 1$ | $Inv_{17} \equiv j < x \wedge j < x - i$ |
| $Inv_{11} \equiv j \neq x \wedge x < 1$ | $Inv_{18} \equiv j \neq x \wedge j \neq x - i$ |
| $Inv_{12} \equiv j \neq x \wedge x \neq 1$ | $Inv_{19} \equiv j > x \wedge x < 1 \wedge j > x - i$ |
| $Inv_{13} \equiv j > x \wedge x < 1 \wedge i > -1$ | |

**Fig. 6.** Application of the algorithm on example Fib. Again, technically, $i$ and $j$ in the open goals are Skolem terms like $f_a(x_a, x_b)$ in Fig. 4. In iteration no. 6, the non-termination proof can be closed with the constraint $[\, x_n < 1 \wedge -2 < x_n \,]$ for the free variables. This result expresses that for $n$ being 0 or $-1$, Fib does not terminate. The following invariants were dismissed by the filters because of inconsistency: $j > x_n \wedge j < 1 + x_n$, $j < x_n \wedge j > x_n - 1$, and $j > x_n \wedge x_n < 1 \wedge j < 1 + x_n$.

(and sometimes the only one) which leads to successful non-termination proofs. Unfortunately, they increase the complexity of proofs in case they do not lead to a closed proofs. In some cases this led to the situation that the algorithm reached the limit of iterations before a suitable invariant was found. This is also the case when the target program is actually terminating for all initial states. The average number of iterations in successful cases (that means a suitable invariant was found) lay between 1.5 and 3.5 depending on the heuristics.

The example Lcm of Sect. 4 was solved in all experiments. The number of necessary iterations lay between 2 and 8 iterations. The example Fib was solved by some of the experiments and their number of iterations was between 6 and 39 iterations. The best run is illustrated in Fig. 6. Using the creation method IneqVar, the number of iteration raises (depending on the heuristics). An invariant which was found in this case is $j > L_j \wedge i > L_i$, where the proof was closed with the (simplified) constraint $[\, L_j = -1 \wedge L_i = -1 \wedge x < 0 \,]$. Invariant and constraint describe the situation where the input value $n$ is negative and the variables $j$ and $i$ are non-negative (which is always the case).

The example set and the implementation of the software is publicly available at `http://www.key-project.org/nonTermination/`.

## 6 Related Work

Although the development of *termination checkers* is a flourishing research subject, we only know of two methods (and implementations) that are directly comparable to the *non-termination* analysis presented in this paper:

The more similar approach is [11], which uses concolic program execution to search for lassos (loops) in a program, and constraint solving for proving the feasibility of lassos. The latter part is similar to the invariant generation method shown in the present paper, but it does not make use of counterexamples to refine invariant candidates. Because we use purely symbolic reasoning to determine critical initial program states, it can also be expected that our approach is able to derive more general descriptions of such input states than [11], at the cost of being less scalable.

Secondly, the AProVE system [12] is able to prove both the termination and non-termination of term rewrite systems [13] and is in principle also applicable to imperative programs: such programs can be analysed after a suitable translation to rewrite systems [2]. So far, existing translations are incomplete, however, which means that the resulting rewrite system might be non-terminating even if the original program is terminating.

Construction of invariants using invariant templates and constraint solving is an approach that is employed in many contexts, e.g., [14, 15]. The principle is usually not embedded in a program logic as it is done in the present paper.

The iterative refinement of invariants described in this paper has some similarities to iterative backwards-propagation of assertions, which is described in [16] but can, in some form or another, be found in many static program analysis techniques.

## 7 Conclusion and Future Work

We have introduced a novel approach to automated detection of non-termination defects in software programs. The approach is built on the basis of a sequent calculus for dynamic logic and works by generating invariants that prove the unreachability of terminating states. In experiments, the majority of our example programs could automatically be proven non-terminating. Furthermore, when experimenting with more complex non-terminating Java programs [5], we found that also here it is often possible to find small and simple invariants that witness non-termination. The intuitive explanation for this is that (i) the usage of the invariant rule While (with anonymising updates) allows to ignore those parts of the program state that are not changed in the loop, and that (ii) the precise character of state changes caused by a loop can be ignored in the invariant as well, as long as non-termination is preserved. Although further investigations concerning such programs are necessary, this indicates that our method is also applicable to programs that operate on heap data structures.

When moving from the while-language to actual Java-like programs, one modification of the algorithm that appears helpful is to automatically add heap-

wellformedness conditions to the invariant candidates. Partly, this is a consequence of using dynamic logic for Java [8, Sect. 3], in which properties like "attributes of allocated objects only point to allocated objects" are non-trivial and can be difficult to synthesise for the invariant generator. Another aspect that becomes more central with Java programs is the detection of the variables and heap locations that a loop can assign to. It might be useful to determine also these locations incrementally and simultaneously with the loop invariant, based on failed proof attempts.

As a prerequisite for more extensive experiments, we want to develop an implementation of our non-termination checker that is more tightly integrated with the program verification tool used. This way, we expect to achieve a significantly higher performance. On the more theoretic level, we are in the process of investigating the usage of closure constraints (Sect. 3) more systematically in order to define fragments of first-order logic with integer arithmetic for which the calculus is complete, and in order to further develop the approach.

## Acknowledgements

We like to thank the following people for constructive feedback and support: Richard Bubel, Prof. Reiner Hähnle, Mattias Ulbrich, Benjamin Weiss and all others of the KeY-group. Besides we like to thank Prof. Giesl for supporting the diploma thesis which was the base for this paper.

## References

1. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: CAV '06, Proceedings of the 18th International Conference on Computer-Aided Verification. LNCS, Springer (2006) 415–418
2. Sondermann, M.: Automatische Terminierungsanalyse von imperativen Programmen. Master's thesis, RWTH Aachen University, Aachen, Germany (2006)
3. Rümmer, P., Shah, M.A.: Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In Gurevich, Y., Meyer, B., eds.: Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland. Revised Papers. Volume 4454 of LNCS., Springer (2007) 41–60
4. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. 2nd edn. Springer-Verlag, New York (1996)
5. Velroyen, H.: Automatic non-termination analysis of imperative programs. Master's thesis, Chalmers University of Technology, Aachen Technical University, Göteborg, Sweden and Aachen, Germany (2007)
6. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge, Massachusetts (1993)
7. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
8. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2007)
9. Rümmer, P.: A sequent calculus for integer arithmetic with counterexample generation. In: Proceedings of 4th International Verification Workshop (VERIFY'07). Volume 259 of CEUR (`http://ceur-ws.org/`). (2007)

10. Giese, M.: Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proceedings, IJCAR, Siena, Italy. Volume 2083 of LNAI., Springer (2001) 545–560
11. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In Necula, G.C., Wadler, P., eds.: ACM Symposium on Principles of Programming Languages (POPL), San Francisco, USA, ACM (2008) 147–158
12. Giesl, J., Schneider-Kamp, P., Thiemann, R.: Aprove 1.2: Automatic termination proofs in the dependency pair framework. In: Proceedings, IJCAR, Seattle, USA. Volume 4130 of LNAI., Springer (2006) 281–286
13. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In Gramlich, B., ed.: FroCos. Volume 3717 of LNCS., Springer (2005) 216–231
14. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In Baader, F., Baumgartner, P., Nieuwenhuis, R., Voronkov, A., eds.: Deduction and Applications. Number 05431 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany (2006)
15. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In Jr., W.A.H., Somenzi, F., eds.: CAV. Volume 2725 of LNCS., Springer (2003) 420–432
16. Bjørner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theor. Comput. Sci. **173**(1) (1997) 49–87