BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SYNTHESIZING NON-TERMINATION PROOFS FROM TEMPLATES

SYNTÉZA DŮKAZŮ NEKONEČNOSTI BĚHU PROGRAMŮ S VYUŽITÍM ŠABLON

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. ŠTEFAN MARTIČEK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Master's Thesis Specification/13436/2016/xmarti62

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems Academic year 2016/2017

Master's Thesis Specification

For:	**Martiček Štefan, Bc.**
Branch of study:	Intelligent Systems
Title:	**Synthesizing Non-Termination Proofs from Templates**
Category:	Formal Verification

Instructions for project work:

1. Study and discuss existing approaches for proving non-termination.
2. Get acquainted with the source code of the 2LS tool and its mechanisms for synthesizing abstractions from templates.
3. Propose a solution for describing recurrence relations for proving non-termination of programs by means of templates, an algorithm for synthesizing recurrence relations from these templates, and a way to extract witnesses for non-terminating executions.
4. Implement the generation of the templates and the synthesis algorithm within the 2LS tool.
5. Evaluate on the benchmarks of the termination category of the Software Verification Competition.
6. Discuss the advantages and limitations of the approach as well as possibilities of its future development.

Basic references:

- Podelski, A., Rybalchenko, A.: Transition Invariants and Transition Predicate Abstraction for Program Termination, In: Proc. of TACAS'11, LNCS 6605, Springer, 2011.
- Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G.: Proving Non-Termination, In: Proc. of POPL'08, ACM Press, 2008.
- Chen, H.-Y., Cook, B., Fuhs, C., Nimkar, K., OHearn, P.: Proving Nontermination via Safety, In: Proc. of TACAS'14, LNCS 8413, Springer, 2014.
- David, C., Kroening, D., Lewis, M.: Unrestricted Termination and Non-termination Arguments for Bit-Vector Programs, In: Proc. of ESOP'15, LNCS 9032, Springer, 2015.

Requirements for the semestral defense:

- The first two items of the subject specification plus at least the design phase from the third item.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Vojnar Tomáš, prof. Ing., Ph.D.**, DITS FIT BUT
Consultant:	Schrammel Peter, UOx
Beginning of work:	November 1, 2016
Date of delivery:	May 24, 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav intelligentních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

Petr Hanáček
Associate Professor and Head of Department

Abstract

One of the properties that are most difficult to verify in the area of formal analysis is liveness. Proving non-termination of programs also belongs to the methods that verify this property. Our work describes the design and implementation of two algorithms checking non-termination. We inspire ourselves by already existing approaches, such as recurrence sets and over-approximation of loops with the use of invariants in the form of recurrence relations. The main challenge for us was an adaptation of these algorithms to the SSA (single static assignment) representation used in 2LS and the overall integration in our framework. We were able to unify the mentioned approaches into analysis of non-termination, which achieves the best results in comparison to the other tools that were compared at the SV-COMP 2017 competition.

Abstrakt

Jednou z nejsložitěji verifikovaných vlastností programů v oblasti formální analýzy je živost. K jedné z metod ověřujících tuto vlastnost patří i dokazování neukončitelnosti programů. Naše práce popisuje návrh a implementaci dvou algoritmů ověřujících neukončitelnost. Inspirujeme se již existujícími přístupy, jako jsou rekurentní množiny a nadaproximace cyklů s využitím invariantů ve tvaru rekurentních relací. Hlavní výzvu pro nás představovalo přizpůsobení těchto algoritmů SSA (single static assignment) reprezentaci použité v 2LS a jejich celková integrace v našem frameworku. Vzpomínané přístupy se nám podařilo spojit do analýzy neukončitelnosti, která dosahuje nejlepší výsledky v porovnání s existujícími nástroji, které byly srovnané na soutěži SV-COMP 2017.

Keywords

termination, non-termination, 2LS, bit-vectors, singleton recurrence set, periodical recurrence set

Klíčová slova

ukončitelnost, neukončitelnost, 2LS, bitové vektory, jednoprvková rekurentní množina, periodická rekurentní množina

Reference

MARTIČEK, Štefan. *Synthesizing Non-Termination Proofs from Templates.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

Synthesizing Non-Termination Proofs from Templates

Declaration

Hereby I declare that this Term project was prepared as an original author's work under the supervision of Prof. Ing. Tomáš Vojnar Ph.D.. The supplementary information was provided by Dipl.-Ing. Dr. Peter Schrammel. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Štefan Martiček
May 24, 2017

Acknowledgements

I would like to thank my supervisor Prof. Ing. Tomáš Vojnar PhD. for his willingness to help, his flexibility and providing advice whenever I needed it. Great thanks belong to my internship mentor Dipl.-Ing. Dr. Peter Schrammel, who provided technical advice and assistance in conducting this thesis. My thanks belongs also to Bc. Viktor Malík, who was very helpful in regards to my understanding of the 2LS tool. Last, but not least, I want to say a special thanks to my friend David Hall FCCA, who helped me with reviewing my use of the English language.

Contents

1 Introduction **3**

2 The 2LS Tool **5**
- 2.1 Abstract Interpretation in 2LS 5
- 2.2 Template-based Approach in 2LS 6
- 2.3 SSA-based Representation 6

3 Termination Analysis **11**
- 3.1 Well-foundedness and Ranking Functions 11
- 3.2 Intra-procedural Termination Analysis in 2LS 11

4 Non-termination Analysis **13**
- 4.1 Lasso-based Approach for Proving Non-termination 13
- 4.2 Bit-level Non-termination Analysis with a Lasso-based Approach 15
- 4.3 Integer Non-termination Analysis with Lasso-based Approach 16
- 4.4 Non-termination Analysis via Safety 16

5 Singleton Recurrence Set Search Algorithm **18**
- 5.1 SSA-based Program Representation for Non-termination Checking 18
- 5.2 SSA-based Non-termination Checking 20

6 Periodical Recurrence Set Search Algorithm **24**
- 6.1 Detection of Linearly Changed Phi Nodes in a Loop 25
- 6.2 Checking Existence of a Periodical Recurrence Set in a Loop 26
- 6.3 Generalization of the Periodical Recurrence Set Search 28
- 6.4 Implementation of Periodical Recurrence Set Search Algorithm 28
  - 6.4.1 Implementation of Detection of Linearly Changed Phi Nodes 29
  - 6.4.2 Implementation of The Non-termination Check 29

7 Implementation **32**

8 Experiments **34**
- 8.1 Benchmarking of the Singleton Recurrence Set Method 35
- 8.2 Benchmarking of the Periodical Recurrence Set Method 37
- 8.3 Comparison with the Other Tools 38
- 8.4 Bug Report 41

9 Conclusion 42

Bibliography 43

Chapter 1

Introduction

Various fully automatic methods of static formal analysis play an increasingly important role in assuring software quality. Nevertheless, this concerns mainly checking relatively simple properties and often using analyses that, in order to gain scalability, give up not only completeness but even soundness. Scalable and sound methods of static analysis of complex properties of real-life programs do still pose a lot of research challenges. This is particularly true for analyses of program termination and non-termination. There exists continuous research in these fields, but the community working on these issues is not as large as in other areas of static analysis, especially in the case of non-termination.

One of the promising tools for formal analysis and verification of C programs is 2LS, supported by the DiffBlue company. The 2LS tool is based on the CPROVER infrastructure. 2LS combines bounded model checking, k-induction and abstract interpretation to implement a new efficient method called k-invariants k-induction [3]. The k-invariants k-induction algorithm uses abstract interpretation to infer inductive invariants. To restrict the space in which possible invariants are to be sought, 2LS uses templates in the forms of parameterized constraints on program variables. For instance, a template for a variable i that has values from the interval $\langle C2, C1\rangle$ is $i \leq C1 \wedge -i \leq C2$. Searching for invariants is then reduced to searching for suitable values of the parameters of the chosen template. In the example, that would mean searching for the values of the parameters $C1$ and $C2$.

2LS also includes a module for verification of termination, based on using lexicographic ranking functions [12]. This termination analysis is currently able to check never-termination (i.e., checking that there does not exist any run from the beginning of a procedure to its end), but the more demanding non-termination analysis, which checks whether there is some non-terminating run of the given procedure, is missing. The goal of this work is to propose, implement, and experimentally evaluate a non-termination analysis in 2LS.

The 2LS tool uses a representation of programs based on the SSA (Single Static Assignment) form, which is the transformation of a program, where every variable is assigned once at most.

The most common technique to check program non-termination is to search for recurrence sets. A recurrence set is a set of program states where at least one state is reachable from the set of initial states, and, for every state in the recurrence set, we are able to get back to the recurrence set by executing the program. In this work, since 2LS uses bit-vectors to represent program states, we use a notion of recurrence sets specialized for the domain of bit-vectors as described by [10]. Hence, we look for one program state that is reachable at some loop head from some initial program state and that can be reached repeatedly from this loop head.

The main challenge that we address is to apply the aforementioned idea on the program representation used in 2LS, which is specific SSA, form encoded as a single bit-vector formula. The SSA form uses phi nodes joining program states, flowing to loop heads from before loops with those flowing back from the loops. The latter states are in 2LS over-approximated by a free variable that can be constrained by gradually refined invariants. These invariants can, however, still over-approximate reachable states. Hence, when using them, we would not be able to prove non-termination. Therefore, we proposed a simple, yet, as our experiments show, quite efficient approach, in which we use incremental loop unwinding, and restricting the SSA form to never use over-approximated back-edges of program loops.

Moreover, we also propose another method for checking non-termination of programs where the above approach tends to be very inefficient. As introduced by [11], we replace the recurrence relations defined by loops with their solutions using loop invariants and thus, we describe arbitrarily many loop iterations in one formula. Contrary to [11], where the loop exit conditions are taken into account, we consider every loop being nonterminating and the loop exit conditions are investigated separately. The effectiveness of this approach lies in the representation of a periodical recurrence set by single formula. A periodical recurrence set is a recurrence set, where the transition from a state in the recurrence set back to the set is defined as an addition of some constant C.

We have implemented and combined both of the approaches in 2LS. We tested our approach on the set of benchmarks used at SV-COMP 2017 (International Competition on Software Verification). 2LS participated in this competition in 2016 and 2017. In 2017, it gained 899 points, which our approach is able to improve to 1489 points. Moreover, our method for checking non-termination performed the best when compared to the non-termination analyses of all other tools that participated in SV-COMP 2017.

The rest of the thesis is structured as follows. Chapter 2 describes the principles of the 2LS tool, mainly abstract interpretation, the template-based approach, and the SSA form used for the program representation. Chapter 3 provides an introduction to intra-procedural termination analysis in 2LS. It introduces the notion of ranking functions and includes some essential definitions. Basic principles of non-termination analysis are presented in Chapter 4. Our method of searching for a singleton recurrence set and its implementation is presented in Chapter 5. We describe the search for periodical recurrence sets and their implementation in Chapter 6. Our experiments done on the set of benchmarks from SV-COMP 2017 are described in Chapter 8. Finally, Chapter 9 gives a summary of our results and contains suggestions for prospective improvements.

Chapter 2

The 2LS Tool

2LS is a static analysis and verification tool for C programs [13]. It performs interprocedural abstract interpretation, verification and refutation of assertions and termination analysis. The tool works on bit-vectors.

2LS uses GOTO programs as an intermediate representation [6]. It performs static analysis to derive the data flow equations for each function of the GOTO program. The result is an SSA form, which is an over-approximation of the GOTO program, because the loops are cut so that the variables modified inside them are havocked at the loop head (see Section 2.3).

For the fast incremental solving 2LS uses SAT solvers, concretely MiniSAT 2.2.1. An SSA equation is translated into a CNF formula by bit-precise modeling of all expressions plus the Boolean guards. The formula is incrementally extended to perform an invariant generation using template-based synthesis, to add further loop unwindings and assertions for property checks. If a property check is satisfiable and the model computed by the SAT solver does not take a path through an invariant, then it corresponds to a path violating at least one of the assertions in the verified program. A human-readable counterexample is provided by translating the model back to a sequence of assignments. An unsatisfiable property check means that the assertions are proven.

The 2LS combines incremental bounded model checking, k-induction and abstract interpretation to create a new k-invariant k-induction algorithm [3]. The content of the following sections is mainly taken from [3]. For the following text we have a notation that $\mathbf{x}$ represents a vector and x scalar.

2.1 Abstract Interpretation in 2LS

One of the main verification methods used in 2LS is an *abstract interpretation* [9]. For the vector of variables $\mathbf{x}$, we describe the start states by the predicate $\mathit{Init}(\mathbf{x})$ and the transition relation by the predicate $\mathit{Trans}(\mathbf{x}, \mathbf{x}')$. The transition relation is a formula which describes the progression relation from one state to another after executing some program statement. We use *inductive invariants* to describe the set of states that is a fixed-point for the transition relation $\mathit{Trans}(\mathbf{x}, \mathbf{x}')$ as follows:

Definition 2.1.1. (Inductive invariant) Inv is an inductive invariant if it has the following property:

$$\forall \mathbf{x}, \mathbf{x}' : \mathit{Inv}(\mathbf{x}) \wedge \mathit{Trans}(\mathbf{x}, \mathbf{x}') \implies \mathit{Inv}(\mathbf{x}') \tag{2.1}$$

The abstract interpretation in 2LS is used to compute inductive invariants $AInv$ which include start states $Init(\mathbf{x})$:

$$\begin{aligned} \exists_2 AInv \in \mathcal{A} : \forall \mathbf{x}, \mathbf{x}' : \quad & (Init(\mathbf{x}) \implies AInv(\mathbf{x})) \wedge \\ & (AInv(\mathbf{x}) \wedge Trans(\mathbf{x}, \mathbf{x}') \implies AInv(\mathbf{x}')) \end{aligned} \tag{2.2}$$

$\mathcal{A}$ is the chosen abstract domain of the formulas. The result of the abstract interpretation is an over-approximation of the set of reachable states. The check for safety is then performed in the following way:

$$\forall \mathbf{x} : AInv(\mathbf{x})) \implies \neg Err(\mathbf{x}) \tag{2.3}$$

If the formula 2.3 does not have a model, this means that the system is safe. Otherwise, we have to find more restrictive $AInv$ or choose the more expressive abstract domain $\mathcal{A}$. In 2LS the template-based approach is used to choose the abstract interpretation domain, i.e. $\mathcal{A}$ contains only the formulas described by a template.

2.2 Template-based Approach in 2LS

In 2LS the state space for the invariants is restricted by a template of the form $\mathcal{T}(\mathbf{x}, \boldsymbol{\delta})$, where $\mathbf{x}$ are program variables and δ are the template parameters. The second-order search for an invariant described by formula (2.2) can thus be replaced by the first-order search for the parameters of the template:

$$\begin{aligned} \exists \boldsymbol{\delta} : \forall \mathbf{x}, \mathbf{x}' : \quad & (Init(\mathbf{x}) \implies \mathcal{T}(\mathbf{x}, \boldsymbol{\delta})) \wedge \\ & (\mathcal{T}(\mathbf{x}, \boldsymbol{\delta}) \wedge Trans(\mathbf{x}, \mathbf{x}') \implies \mathcal{T}(\mathbf{x}', \boldsymbol{\delta})) \end{aligned} \tag{2.4}$$

The quantifier alternation $\exists\forall$ in the formula (2.4) is a challenge for today's SMT solvers. Therefore the formula is negated and the parameters $\boldsymbol{\delta}$ are searched iteratively for different choices of constants $\mathbf{d}$:

$$\begin{aligned} \exists \mathbf{x}, \mathbf{x}' : \quad & \neg(Init(\mathbf{x}) \implies \mathcal{T}(\mathbf{x}, \mathbf{d})) \vee \\ & \neg(\mathcal{T}(\mathbf{x}, \mathbf{d}) \wedge Trans(\mathbf{x}, \mathbf{x}') \implies \mathcal{T}(\mathbf{x}', \mathbf{d})) \end{aligned} \tag{2.5}$$

The abstract value $\mathbf{d}$ represents the set of all $\mathbf{x}$ that satisfy the formula $\mathcal{T}(\mathbf{x}, \mathbf{d})$. The abstract value $\perp$ denotes empty set and $\top$ stands for the whole domain. Hence we get $\mathcal{T}(\mathbf{x}, \perp) \equiv false$ and $\mathcal{T}(\mathbf{x}, \top) \equiv false$.

2.3 SSA-based Representation

A program verified by 2LS is at first translated into GOTO representation. In the GOTO version of a program, loop heads have a form of conditions and goto statements. The GOTO representation is then converted to SSA-based representation, which will be further called the SSA form [7]. We explain our approach on a simple program below. We further provide the corresponding GOTO program and the SSA form. We also employ CFG (Control Flow Graph) to explain the use of the specific constructs of the SSA form.

Listing 2.1: The example part of a program in C

```
int i = 1;
while (i > 0)
  i--;
return 0;
```

Listing 2.2: The corresponding GOTO program to the code 2.1

```
    signed int i;
    i = 1;
1:  IF !(i >= 1) THEN GOTO 2
    i = -1 + i;
    GOTO 1
2:  return_value = 0;
    dead i;
```

Listing 2.3: The SSA form of the code 2.1

```
$guard#0 == TRUE
i#1 == 1
i#phi2 == ($guard#ls4 ? i#lb4 : i#1)
$cond#2 == !(i#phi2 >= 1)
$guard#2 == $guard#0
i#3 == -1 + i#phi2
$guard#3 == (!$cond#2 && $guard#2)
$cond#4 == TRUE
main#return_value#5 == 0
$guard#5 == ($cond#2 && $guard#2)
```

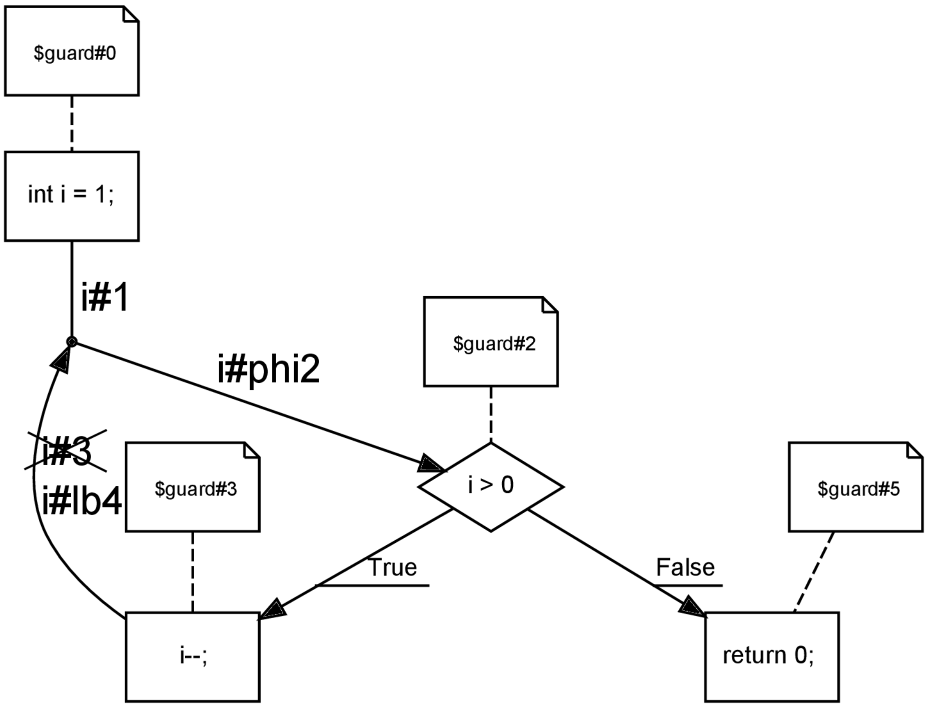

Figure 2.1: CFG with the corresponding parts of the SSA form

The parts of the generated SSA from 2.3 correspond to GOTO program statements. The main difference is the use of guard variables, which encode reachability of some program locations (see Figure 2.1). `$guard#0` represents the reachability of the function entry point. It is set to `TRUE`, because we do intra-procedural analysis. `$guard#2` represents the reachability of the loop head. `$guard#3` holds when the reachable loop has the satisfiable condition `$cond#2`. Because of the different types of loops in C there are 2 conditions for every loop in the SSA form. `$cond#2` represents a condition at the loop head (for, while) and `$cond#4` is a condition at the loop end (do-while). The last guard - `$guard#5` holds when the location after the loop is reached. According to the equation in the SSA form in Listing 2.3, this happens when the loop condition is not satisfied and the loop is reachable.

Notice the equation `i#phi2 == ($guard#ls4 ? i#lb4 : i#1)`, where we use instead of `i#3` a free variable `i#lb4`. This is shown in CFG in Figure 2.1 where we cut the back edge variable `i#3` and replace it with `i#lb4`. Even though the CFG still depicts a loop, there is no dependency between `i#3` and `i#lb4`. The latter is a free variable representing an abstraction. Also, instead of a guard at the loop back, which is not even created (it must encompass `$cond#4`), we use another free variable `$guard#ls4`. This is an abstraction that covers the effect of any number of loop iterations because `i#lb4` is a free variable and the choice between `i#lb4` and `i#1` is nondeterministic. As you can see, there is no loop in the generated SSA, and so it became an acyclic over-approximation of the program. If the property holds for both `i#1` and `i#3`, it can be assumed to hold for `i#lb4`. An invariant is computed in the form of constraints for `i#lb4`.

The way we can make the analysis more precise is to employ a loop unwinding. It is performed by repeating the conversion of a loop and renaming the variables. The variables from different loop exits are then merged at the end. The separate loop bodies created from one loop are called *loop unwindings*. Guard variables of loop heads of different loop unwindings are used as conditions for variable selection. We will demonstrate the unwinding procedure on the same program as we used above in Listing 2.1. The following is the unwound program in C:

Listing 2.4: The example part of a program in C

```
int i = 1;
while (i > 0)
{
  i--;
  if (!(i > 0)) goto BREAK;
  i--;
}
BREAK:
return 0;
```

The corresponding SSA form becomes complicated and hardly readable as shown in Listing 2.5.

Listing 2.5: The SSA form of the code from Listing 2.1 with every loop unwound one time

```
$guard#0 == TRUE
i#1 == 1
i#phi2%1 == ($guard#ls4%1 ? i#lb4%1 : i#1)
$cond#2%0 == !(i#phi2%0 >= 1)
$guard#2%0 == ($guard#3%1 && $cond#4%1)
$cond#2%1 == !(i#phi2%1 >= 1)
$guard#2%1 == $guard#0
i#3%1 == -1 + i#phi2%1
i#phi2%0 == i#3%1
```

```
i#3%0 == -1 + i#phi2%0
$guard#3%1 == (!$cond#2%1 && $guard#2%1)
$guard#3%0 == (!$cond#2%0 && $guard#2%0)
$cond#4%1 == TRUE
$cond#4%0 == TRUE
$cond#2 == ($guard#2%1 && $cond#2%1 ? $cond#2%1 : $cond#2%0)
$guard#2 == ($guard#2%1 && $cond#2%1 ? $guard#2%1 : $guard#2%0)
i#phi2 == ($guard#2%1 && $cond#2%1 ? i#phi2%1 : i#phi2%0)
main#return_value#5 == 0
$guard#5 == ($cond#2 && $guard#2)
```

We add the corresponding CFG in Figure 2.2 which assigns guards to program locations.

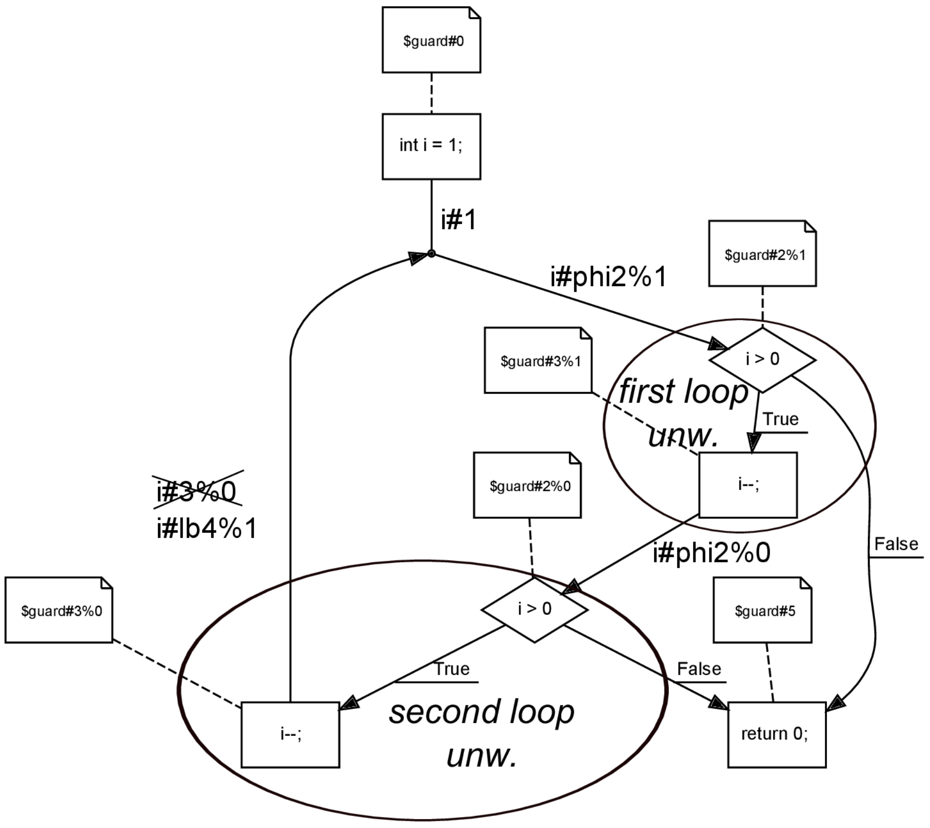

Figure 2.2: CFG with the corresponding parts of the SSA form

We can see some already known parts that were described in Listing 2.3. To differentiate two loop unwindings of the same loop body, 2LS uses suffix `%i`. It is important to mention that `%1` denotes the first iteration and `%0` denotes the second. The loop unwinding that gets the highest number in its suffix always represents the first loop iteration. Contrariwise, an instance of the loop body with the suffix `%0` is always the last instance.

The unwound version has some additional program locations. As we can see different instances of the loop are bound together by the equations `i#phi2%0 == i#3%1` and `$guard#2%0 == ($guard#3%1 && $cond#4%1)`. The first equation binds the output variable of the first instance to the input variable of the second one. We can see in the second equation that `$guard#2%0`, which represents the location of the second loop unwinding, holds only when the loop body of the previous instance has been entered and the loop end condition of the first unwinding holds. Results from different loop instances are merged together and suffix `%i` is removed. Thus the loop result can be later used as if there was only one loop instance - `$guard#5 == ($cond#2 && $guard#2)`.

Chapter 3

Termination Analysis

This chapter is devoted to the description of termination analysis implemented in 2LS. It was the stepping stone of our further studies. The non-termination analysis itself would be useless without integration of termination analysis into the cooperating model. Therefore, we describe the theoretical principles of the algorithm implemented in 2LS checking termination. 2LS contains the implementation of intra-procedural as well as inter-procedural analysis. Inter-procedural aspect is not relevant here and will not be presented, because it was not used in our work. Our methods and also termination analysis used in this thesis are intra-procedural. It is ensured by inlining all function calls, and as we know, recursion in 2LS is not supported. The content of the following sections is predominantly drawn from [5].

3.1 Well-foundedness and Ranking Functions

Definition 3.1.1. (Well-founded Relation [12]) A relation $R \subseteq X \times X$ is well-founded iff every non-empty subset of X has an R-minimal element.

Lemma 1. A binary relation R is well-founded if and only if there exists a ranking function for R.

Definitions of Ranking function 3.1.2 and the following lemma are taken from [8].

Definition 3.1.2. (Ranking Function) Suppose (D, $\prec$) is a well-founded, strictly partially ordered set, and $R \subseteq U \times U$ is a relation over a non-empty set U. A ranking function for R is a function $m : U \to D$ such that:

$$\forall a, b \in U : R(a, b) \implies m(b) \prec m(a) \tag{3.1}$$

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

3.2 Intra-procedural Termination Analysis in 2LS

The linear lexicographic ranking functions are used in 2LS to efficiently solve the termination problem, because monolithic ranking functions, however they are complete, are much more difficult to solve by existing SMT solvers.

Definition 3.2.1. (Lexicographic Ranking Function) A lexicographic ranking function R for a transition relation $\mathit{Trans}(\mathbf{x}, \mathbf{x}')$ is an n-tuple of expressions $(R_n, R_{n-1}, ..., R_1)$ such that

$$\exists \Delta > 0 : \forall \mathbf{x}, \mathbf{x}' : \mathit{Trans}(\mathbf{x}, \mathbf{x}') \wedge \exists i \in [1, n] : \begin{array}{l} R_i(\mathbf{x}) > 0 \\ \wedge R_i(\mathbf{x}) - R_i(\mathbf{x}') > \Delta \\ \wedge \forall j > i : R_j(\mathbf{x}) - R_j(\mathbf{x}') \geq 0 \end{array} \tag{3.2}$$

The existence of $\Delta > 0$ and the condition $R_i(\mathbf{x}) > 0$ guarantee that the relation $>$ is well-founded. Since 2LS works on bit-vectors, the condition $R_i(\mathbf{x}) > 0$ is trivially satisfied. Bit-vectors are also discrete so we replace the condition $R_i(\mathbf{x}) - R_i(\mathbf{x}') > \Delta$ with $R_i(\mathbf{x}) - R_i(\mathbf{x}') > 0$. The condition that $(R_n, R_{n-1}, ..., R_1)$ is a lexicographic ranking function with n components over bit-vectors is necessary and sufficient for the validity of the following LR formula:

$$LR^n(\mathbf{x}, \mathbf{x}') = \bigvee_{i=1}^{n} \Big(R_i(\mathbf{x}) - R_i(\mathbf{x}') > 0 \wedge \bigwedge_{j=i+1}^{n} (R_j(\mathbf{x}) - R_j(\mathbf{x}') \geq 0) \Big) \tag{3.3}$$

The procedure f may be composed of several loops, where each of the loops has a guard g that expresses the reachability of the loop head. For k loops in the procedure f the lexicographic ranking function has the form:

$$RR^n(\mathbf{x}, \mathbf{x}') = \bigwedge_{i=1}^{k} g_i(\mathbf{x}) \implies LR^n(\mathbf{x}, \mathbf{x}') \tag{3.4}$$

To synthesize lexicographic ranking functions the function $R_i(\mathbf{x})$ is specified to be the product $\mathbf{l_i x}$ where l_i is the template parameter vector. The resulting constraints for a loop i are $\mathcal{LR}_i^{n_i}(\mathbf{x}, \mathbf{x}', \mathbf{L_i^{n_i}})$, where $\mathbf{L_i^{n_i}}$ is the vector $(\mathbf{l_i^1}, ..., \mathbf{l_i^{n_i}})$. The constraints for the whole procedure are $\mathcal{RR}(\mathbf{x}, \mathbf{x}', \mathbf{L^n})$, where $\mathbf{L^n}$ is the vector $\mathbf{L_1^{n_1}}, ..., \mathbf{L_k^{n_k}}$. When no ranking function has been found, the special value $\top$ is used. The initial value of the template is $\bot$ which means that the ranking function has not been computed yet. Using these values we get $\mathcal{LR}_i^{n_i}(\mathbf{x}, \mathbf{x}', \top) \equiv \mathit{true}$ and $\mathcal{LR}_i^{n_i}(\mathbf{x}, \mathbf{x}', \bot) \equiv \mathit{false}$. Finally, the reduction of the ranking function synthesis to a first-order quantifier elimination problem over templates is Formula 3.5:

$$\exists \mathbf{L^n} : \forall \mathbf{x}, \mathbf{x}' : \mathit{Inv}(\mathbf{x}) \wedge \mathit{Trans}(\mathbf{x}, \mathbf{x}') \implies \mathcal{RR}(\mathbf{x}, \mathbf{x}', \mathbf{L^n}) \tag{3.5}$$

Chapter 4

Non-termination Analysis

Since the termination analysis in 2LS computes only the sufficient termination precondition, which is a subset of the weakest precondition, the negation of this precondition could still encompass a terminating path. Therefore, we need to find another way to prove non-termination than just negating the termination analysis results. In this chapter we look at the most commonly used approach to prove non-termination, which are the recurrence sets as described by [4, 10]. Informally a recurrence set *RSet* is a set of states at the head of a loop that satisfies the following properties [10]:

1. *RSet* entails the loop condition.
2. Some reachable state $\mathbf{x}$ satisfies *RSet*.
3. For every state $\mathbf{x}$ satisfying *RSet*, some successor of $\mathbf{x}$, after executing the loop body, is again in *RSet*.

We introduce a theoretical principles of the lasso-based approach mentioned in [10]. And finally we provide a short description of the method that transforms a property check from liveness to safety.

4.1 Lasso-based Approach for Proving Non-termination

A method working on lassos is described in [10]. The lasso consists of a finite program path called *stem* followed by a finite program path named *loop*. The loop must form a syntactic cycle in the control-flow or call graph of the program. If the stem can be followed by an infinite number of executions of the loop, the lasso is feasible.

The method works in two phases. In the first phase lassos are generated. The second phase then checks the feasibility of a given lasso.

Definition 4.1.1. (Feasibility of the Lasso) A lasso is feasible if and only if there exists a recurrence set of states visited infinitely often along the infinite path that results from unrolling the lasso.

Using the convention from Chapter 3, we define the relations $Trans_{stem}$ and $Trans_{loop}$ on the program states as follows:

$$Trans_{stem}(\mathbf{x}_0, \mathbf{x}_k) = Init(\mathbf{x}_0) \wedge \bigwedge_{i=0}^{k-1} Trans(\mathbf{x}_i, \mathbf{x}_{i+1}) \tag{4.1}$$

$$\mathit{Trans}_{loop}(\mathbf{x}_k, \mathbf{x}_m) = \bigwedge_{i=k}^{m-1} \mathit{Trans}(\mathbf{x}_i, \mathbf{x}_{i+1}) \text{ where } m > k \tag{4.2}$$

With the previous representation of the stem and loop transitions, we will now specify the notions of open and closed recurrence sets according to the definitions of [4]. We consider it necessary to give the exact definition of the recurrence set:

Definition 4.1.2. (Open Recurrence Set) A transition relation Trans_{loop} with initial states Init has an *(open) recurrence set* of states RSet iff Formulas 4.3 hold.

$$\begin{aligned} &\exists \mathbf{x}, \mathbf{x}' : \mathit{Trans}_{stem}(\mathbf{x}, \mathbf{x}') \wedge \mathit{RSet}(\mathbf{x}') \\ &\forall \mathbf{x} \exists \mathbf{x}' : \mathit{RSet}(\mathbf{x}) \implies \mathit{Trans}_{loop}(\mathbf{x}, \mathbf{x}') \wedge \mathit{RSet}(\mathbf{x}') \end{aligned} \tag{4.3}$$

Definition 4.1.3. (Closed Recurrence Set) A set RSet is a *closed recurrence set* for a transition relation Trans_{loop} with initial states Init iff the Formulas 4.4 hold.

$$\begin{aligned} &\exists \mathbf{x}, \mathbf{x}' : \mathit{Trans}_{stem}(\mathbf{x}, \mathbf{x}') \wedge \mathit{RSet}(\mathbf{x}') \\ &\forall \mathbf{x} \exists \mathbf{x}' : \mathit{RSet}(\mathbf{x}) \implies \mathit{Trans}_{loop}(\mathbf{x}, \mathbf{x}') \\ &\forall \mathbf{x} \forall \mathbf{x}' : \mathit{RSet}(\mathbf{x}) \wedge \mathit{Trans}_{loop}(\mathbf{x}, \mathbf{x}') \implies \mathit{RSet}(\mathbf{x}') \end{aligned} \tag{4.4}$$

Unlike the open recurrence set, the closed recurrence set requires that every successor of the state in RSet must be the member of RSet. The following theorems make clearer the relationship between a recurrence set and closed recurrence sets.

Theorem 3. (Closed Recurrence Sets are Recurrence Sets [4]) Let RSet be a closed recurrence set for Trans_{loop} with initial states Init. Then RSet is also an open recurrence set for RSet with initial states Init.

Theorem 4. (Open Recurrence Sets Always Contain Closed Recurrence Sets [4]) There exists a recurrence set RSet for a transition relation Trans_{loop} with initial states Init iff there exists an under-approximation Trans'_{loop} with initial states Init' and $\mathit{RSet}' \subseteq \mathit{RSet}$ such that RSet' is a closed recurrence set for Trans'_{loop} with initial states Init'.

Regarding the algorithm, the search for a lasso consists of two phases. In the first phase, the CFG (control flow graph) is searched for a lasso. In the next step, the lasso is checked for non-termination. The whole process is nondeterministic (i.e. each next program state is chosen randomly from the set of accessible states, and when there are more loops in the CFG, the algorithm randomly selects one of them). Backtracking is used to pick all feasible lassos.

Recall that 2LS is acyclic, so we cannot search for lassos in our method, thus instead we check all loops in a program for non-termination. The most important part of the lasso-based approach for our analysis is checking non-termination of the loop, which is discussed below.

Definition 3.1.1 can be used to define non-well-foundedness. We use the informal definition of [10] - the relation $\mathit{Trans}(\mathbf{x}, \mathbf{x}')$ over the program states is not-well-founded if it induces an infinite sequence of states. The goal of this analysis is to find the initial states of such sequences.

Definition 4.1.4. (Infinite Execution of the Lasso) A lasso induces an infinite execution if the following relation is not-well-founded:

$$\exists \mathbf{x}, \mathbf{x}', \mathbf{x}'' : \mathit{Trans}_{stem}(\mathbf{x}, \mathbf{x}') \wedge \mathit{Trans}_{loop}(\mathbf{x}', \mathbf{x}'') \tag{4.5}$$

Proposition 1 taken from [10] relates the property non-well-foundedness and the recurrence sets.

Proposition 1. (Non-well-foundedness and Recurrence Sets) A relation $\mathit{Trans}_{loop}(\mathbf{x}, \mathbf{x}')$ is not-well-founded if and only if there exists a non-empty recurrence set of states, i.e., if for some $\mathit{RSet}(\mathbf{x})$, we have:

$$\begin{aligned} &\exists \mathbf{x} : \mathit{RSet}(\mathbf{x}) \\ &\forall \mathbf{x} \exists \mathbf{x}' : \mathit{RSet}(\mathbf{x}) \implies \mathit{Trans}_{loop}(\mathbf{x}, \mathbf{x}') \wedge \mathit{RSet}(\mathbf{x}') \end{aligned} \tag{4.6}$$

And finally we define the infinite lasso execution using the recurrence sets by the following proposition taken from [10].

Proposition 2. (Non-well-foundedness and Recurrence Sets) A lasso induces an infinite execution if and only if there exists a recurrence set $\mathit{RSet}(\mathbf{x}')$ for the relation $\mathit{Trans}_{loop}(\mathbf{x}', \mathbf{x}'')$ such that:

$$\exists \mathbf{x} \exists \mathbf{x}' : \mathit{Trans}_{stem}(\mathbf{x}, \mathbf{x}') \wedge \mathit{RSet}(\mathbf{x}') \tag{4.7}$$

4.2 Bit-level Non-termination Analysis with a Lasso-based Approach

This analysis represents the core of our work since it works on bit-vectors such as the whole 2LS tool. It assumes finite state space, and thus for an infinite program execution, there exists a state which is repeated an infinite number of times. Therefore we look for a recurrence set with exactly one state. This can be formally expressed by the formula:

$$\exists \mathbf{x}, \mathbf{x}', \mathbf{x}'' : \mathit{Trans}_{stem}(\mathbf{x}, \mathbf{x}') \wedge \mathit{Trans}_{loop}(\mathbf{x}', \mathbf{x}'') \wedge (\mathbf{x}' = \mathbf{x}'') \tag{4.8}$$

In Formula 4.8 Trans_{stem} represents a transition from the initial states to a state at the loop head of a lasso and Trans_{loop} is the predicate for the execution of the loop body.

In some cases we need to apply loop unwinding to find a singleton recurrence set as demonstrated by the following example:

```
while (x==y) {x = !x; y = !y; }
```

The program does not terminate if x equals y at the beginning. By applying Formula 4.8 we get the expression $(x = y) \wedge (x = \neg x \wedge y = \neg y)$, which is unsatisfiable. We need to unwind the loop once to satisfy the formula for the singleton recurrence set:

```
while (x == y) {x = !x; y = !y; if (x==y) {x = !x; y = !y;}}
```

Now we get $(x = y) \wedge (\neg x = \neg y) \wedge (x = \neg\neg x \wedge y = \neg\neg y)$ which is a satisfiable formula.

This method does not require us to solve the problem of quantifier alternation. It simply allows us to use a SAT solver and can be very effective with some program representations as shown by our implementation in 2LS (see Section 5.2). However, for some program instances this might be a very ineffective solution even though they are trivial. For example a simple program:

```
for (int i; 1; i++);
```

will take as many unwindings as is the size of the integer type in C to satisfy Formula 4.8. We later introduce a method that is able to handle a small class of such program instances.

4.3 Integer Non-termination Analysis with Lasso-based Approach

The program transitions can be represented by linear inequalities. The constraint-based approach has been used by [10] to synthesize recurrence sets. The authors have chosen templates consisting of linear inequalities that describe a transition relation and a recurrence set (see Equations 4.9).

$$\begin{aligned} RSet &= T\mathbf{x} \leq t \\ loopguard &= G\mathbf{x} \leq g \\ \mathbf{x}' &= U\mathbf{x} + u \end{aligned} \tag{4.9}$$

By using the template $\mathbf{x}' = U\mathbf{x} + u$ as a substitution for $\mathbf{x}'$ in formula 4.6, the authors moved existential quantification from the next program state to the template parameters:

$$\forall \mathbf{x} : RSet(\mathbf{x}) \implies Trans(\mathbf{x}, U\mathbf{x} + u) \wedge RSet(U\mathbf{x} + u) \tag{4.10}$$

However, there is still a universal quantifier in the formula. To avoid this, authors utilize Lemma 5.

Lemma 5. (Farkas' Lemma) A satisfiable system of linear inequalities $Ax \leq b$ implies an inequality $cx \leq \delta$ if and only if there exists a non-negative vector λ such that $\lambda A = c$ and $\lambda b \leq \delta$.

After the use of Templates 4.9 for Formula 4.10 we get all parameters we need to utilize the Lemma 5.

$$\forall \mathbf{x} : T\mathbf{x} \leq t \implies G\mathbf{x} \leq g \wedge TU\mathbf{x} \leq t - Tu \tag{4.11}$$

In the general case we have more such implications as described by the previous lemma and instead of a search for vector λ it is the search for matrix Λ. In this case the employment of Lemma 5 to Formula 4.11 produces the following formula:

$$\exists \Lambda \geq 0 : \Lambda T = \begin{pmatrix} G \\ TU \end{pmatrix} \wedge \Lambda t \leq \begin{pmatrix} g \\ t - Tu \end{pmatrix} \tag{4.12}$$

A similar approach can be used to transform Formula 4.7. The solution for these constraints gives us a recurrence set. This would be an elegant and simple solution for some class of non-terminating loops. Unfortunately Farkas' lemma is applicable for integers, but there is no such approach to handle quantifier alternation with bit-vectors.

4.4 Non-termination Analysis via Safety

Another approach based on recurrence sets is designed by [4]. Authors apply notion of closed recurrence sets in the form of under-approximation of an existing program using assumptions and assertions. Thus checking a liveness property is transformed into checking a safety property, which is that a program never terminates.

The principle of the method is as follows. We put an assumption `assume(true)` at the beginning of a program. Such assumptions are also placed after each use of nondeterministic values. An `assert(false)` statement is put in every place where a loop may exit. This program transformation is illustrated in Figure 4.1 taken from [4].

```
if (k >= 0)
  skip;
else
  i = -1;
while (i >= 0) {
  i = nondet();
}
i = 2;
```

```
assume(true);
if(k >= 0)
  skip;
else
  i = -1;
while (i >= 0) {
  i = nondet();
  assume(true);
}
assert(false);
i = 2;
```

Figure 4.1: Program transformation

As we can see, every path that violates the added assertion is terminating. The task of this method is to find the restrictive conditions for these assumptions so that no assertion is violated (see Figure 4.2).

```
assume(k >= 0 && i >= 0);
if(k >= 0)
  skip;
else
  i = -1;
while (i >= 0) {
  i = nondet();
  assume(i >= 0);
}
assert(false);
i = 2;
```

Figure 4.2: Program with restrictions specifying the closed recurrence set

The restrictive conditions specify a closed recurrence set as soon as no assertion can be violated. To ensure soundness, the loop reachability is checked after computing assumptions, which is transformed to a safety check, by adding `assert(false)` before the loop. Finally, assertions inside the loop must be checked for satisfiability.

Authors use precondition computation to define error states that violate the added assertions. A crucial part of this method is to use an accurate approach that will not lead to divergence or to an empty recurrence set. The restrictive conditions are computed iteratively, where after computing a precondition, refinement is applied to a condition.

A great advantage of this method is the reduction of liveness to safety, which creates the possibility to use existing effective methods to prove safety properties. However, there is still many programs where it will diverge or fail, because of insufficient or undue refinement.

Chapter 5

Singleton Recurrence Set Search Algorithm

In Chapter 4, we have presented the recurrence set method for proving non-termination of programs. The description was based on an abstract representation of the behavior of programs via the $\mathit{Trans}(\mathbf{x}, \mathbf{x}')$ relation. To recall, this relation represents any single step between any two consecutive program locations in any run of the program. Note that even the program counter is included among the variables in $\mathbf{x}$. A sequence of n steps of the program is then represented using the conjunction $\bigwedge_{i=0}^{n} \mathit{Trans}(\mathbf{x}_i, \mathbf{x}_{i+1})$ with a fresh copy of each program variable for each step.

In what follows, we use a different representation of the behavior of programs that is based on the SSA representation of programs heavily exploited in 2LS. Remember that the SSA representation, common in 2LS, over-approximates the behavior of programs by using a phi node at each loop head, which non-deterministically chooses (using a free guard variable) between values flowing into the loop and values flowing through the back edge (abstracted away using another free variable). This way, 2LS covers the effect of any number of loop iterations, but even if loop invariants are introduced to reduce the non-determinism, some over-approximation typically happens. We cannot afford to have this over-approximation and be able to reliably detect non-termination, and so we will prohibit execution of the back edges in the following, and instead, we will use (exact) loop unwinding.

Below, we first explain our representation of program execution in more detail using an example. We will also explain in a closer way the correspondence between this representation and the representation based on the *Trans* relation, providing a basis for understanding the correctness of our construction. Subsequently, we propose a way that our program representation can be used for non-termination checking.

5.1 SSA-based Program Representation for Non-termination Checking

In order to explain our SSA-based program representation suitable for non-termination checking, we use the following example:

```
l1: int i = 1;
l2: while (i < 10)
l3:   i++;
l4: return 0;
```

The transition relation *Trans* for this program looks as follows:

$$
\begin{aligned}
\mathit{Trans}(\mathbf{x},\mathbf{x}') \equiv\ & \mathbf{x} = (\text{init}, i) \wedge \mathbf{x}' = (\text{l1}, i) \vee \\
& \mathbf{x} = (\text{l1}, i) \wedge \mathbf{x}' = (\text{l2}, 1) \vee \\
& \mathbf{x} = (\text{l2}, i) \wedge i < 10 \wedge \mathbf{x}' = (\text{l3}, i) \vee \\
& \mathbf{x} = (\text{l2}, i) \wedge i >= 10 \wedge \mathbf{x}' = (\text{l4}, i) \vee \\
& \mathbf{x} = (\text{l3}, i) \wedge \mathbf{x}' = (\text{l2}, i+1) \vee \\
& \mathbf{x} = (\text{l4}, i) \wedge \mathbf{x}' = ((\text{end}, 0), 1),
\end{aligned}
\tag{5.1}
$$

On the other hand, the corresponding SSA form without unwinding is:

```
$guard#0 == TRUE
i#1 == 1
$guard#2 == $guard#0
i#phi2 == ($guard#ls4 ? i#lb4 : i#1)
$cond#2 == i#phi2 >= 10
$guard#3 == (!$cond#2 && $guard#2)
i#3 == 1 + i#phi2
$guard#5 == ($cond#2 && $guard#2)
main#return_value#5 == 0
```

The variable `$guard#ls4` represents a nondeterministic choice between the back edge variable `i#lb4` and the incoming variable `i#1`. By this over-approximation we cover an arbitrary number of loop iterations, but we include the loop results that are not feasible. Now let the free variable `$guard#ls4` be 0. So instead of the equation `i#phi2 == ($guard#ls4 ? i#lb4 : i#1)` we get `i#phi2 == i#1`. This will completely destroy a loop and the SSA will only contain the information about its first iteration. Instead of over-approximation, now we get an under-approximation of a program. We will use loop unwinding as described in Section 2.3 to obtain more precise program behavior.

The program without unwinding corresponds to the *Trans* relation applied four times on the initial state (init, i). We get the same result for the variable `i` that we obtain from the solver for the variable `i#3`. Every state reached by applying the *Trans* relation, up to four times is encoded in the SSA form. The initial state (init, i) is the very beginning. `$guard#0 == TRUE` represents the state $(\text{l1}, i)$. By adding the line `i#1 == 1` we get a representation of the state $(\text{l2}, 1)$. The state $(\text{l3}, i)$ is represented by all the SSA equations except the last three. Finally, if we add `i#3 == 1 + i#phi2` we get a representation of the state $(\text{l2}, i+1)$. The value of `i#3` is not passed back to the loop head, hence the information about the loop is lost here.

Let us now take the simplified SSA form, without guards and conditions where the loop is unwound once:

```
i#1 == 1
i#phi2%1 == (0 ? i#lb4%1 : i#1)
i#3%1 == 1 + i#phi2%1
i#phi2%0 == i#3%1
i#3%0 == 1 + i#phi2%0
```

We see that the value of the variable `i#3%0` is equivalent to the value of the variable `i` after *Trans* is applied 6 times. We continue unwinding until we have 10 copies of the loop.

Now, the SSA form unwound 9 times encodes all the states that are reachable by the *Trans* relation. In this case, the SSA form represents an n-ary relation on program states, since it encodes every state along the execution path by a separate equation. Here, we have to consider that this representation is not equivalent to $\bigwedge_{i=0}^{n} \mathit{Trans}(\mathbf{x}_i, \mathbf{x}_{i+1})$. The reason is that the *Trans* relation and the SSA form do not use common variables.

As it was demonstrated in our example, an arbitrary state reachable by *Trans* has its representation in the SSA form unwound sufficiently many times. On the other hand, the SSA form unwound k times represents some finite number n of iterations of the *Trans* relation. A formal proof of this would be rather technical and long. We consider it to be beyond the scope of this thesis, and we suppose that its presence is not necessary to believe it.

The formula $\bigwedge_{i=0}^{n} \mathit{Trans}(\mathbf{x}_i, \mathbf{x}_{i+1})$ is further represented by the formula SSA_k (the SSA form unwound k times). Note that the relationship between these parameters is exponential if we consider the unwinding of nested loops. Let us take this simple program:

```
l1: int i = 1;
l2: while (i < 10)
l3:   while (i < 10)
l4:     i++;
l5: return 0;
```

The unwinding procedure will at first create k loop bodies of the inner loop and then k loop bodies of the outer loop, so at the end, the number of the inner loop bodies is k^2.

5.2 SSA-based Non-termination Checking

Now, we will utilize the representation presented in the previous section to describe our method for non-termination checking. We iteratively employ loop unwinding and we compare the new state we get with the states in the loop heads reached previously. The state in this section does not encompass the program location, since we use the SSA representation. Also remember that we must avoid over-approximation using the back edges here. This simple check is depicted in Figure 5.1.

We will use $\%i$ as a suffix for the symbols from a specific loop unwinding. *loop* is a set of all variables of every unwinding in a loop. *loop_guard*%0 represents the guard of a loop head from the last loop body created by unwinding.

The solver is implemented as a stack. It means that we can push and pop some formulas to and from the solver depending on our needs. As a base for our solution, the solver must always contain the SSA formula updated according to the current unwinding k. Since we do not use over-approximation in our method, we need to add some restrictions to the SSA formula. As we know from Section 2.3, what creates this over-approximation are free variables that replace the back edge variables and the special guards that serve for nondeterministic choice in phi node between incoming variables and back edge variables. Therefore, we create a conjunction of the negations of all these guards, which means that the back edge variables are completely omitted and we under-approximate the program behavior:

$$\bigwedge\nolimits_{loop \in LOOPS} \neg guard\_ls_{loop} \tag{5.2}$$

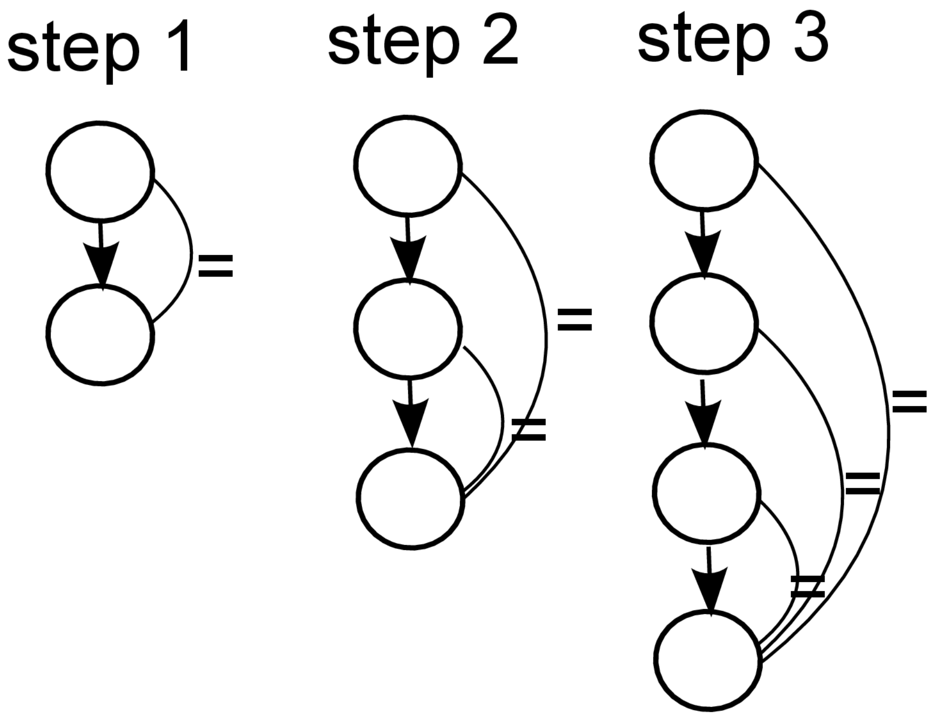

Figure 5.1: Illustration of the method

Now, we describe a simple phi node comparison which represents the search for a singleton recurrence set. We compare variables from different loop unwindings in order to detect a state repetition in the loop. It is important to mention that we compare only the variables modified in the loop, the so called phi nodes of the loop. As our method proceeds iteratively, we make comparison only for the last loop iteration in every step (see Figure 5.1). Other iterations have already been compared to each other in the previous steps. To ensure the feasibility of the verified loop unwinding, we must also add a loop guard of the last iteration into the formula. Below, we introduce a singleton recurrence set check for one specific loop unwinding:

$$loop\_guard\%0 \wedge \bigwedge_{phi\_var\%i \in loop} phi\_var\%i = phi\_var\%0 \tag{5.3}$$

Finally, we put all previously described pieces of our algorithm together and create a complex formula to check one iteration of every loop in the C program. The formula looks as follows:

$$\begin{gathered} SSA_k \wedge \bigwedge_{loop \in LOOPS} \neg guard\_ls_{loop} \wedge \\ \bigvee_{loop \in LOOPS} \bigvee_{i=1}^{k} \Big(loop\_guard\%0 \wedge \bigwedge_{phi\_var\%i \in loop} phi\_var\%i = phi\_var\%0 \Big) \end{gathered} \tag{5.4}$$

This formula is fed into the solver at every step of our algorithm, and, if satisfiable, it is sufficient to prove the existence of a non-terminating program execution. Recall, that it is not equivalent to Formula 4.8. They are not equivalent, because the SSA formula does not use a fresh copy of all variables at each step and also the unwinding procedure may create branches which are not reachable by iteration of the *Trans* relation. However, the formulas are indeed equisatisfiable as we showed in the example in Section 5.1, that we can create a mapping between the states reachable by the *Trans* relation and the states expressed by the formula $SSA_k \wedge \bigwedge_{loop \in LOOPS} \neg guard\_ls_{loop}$ (SSA unwound k times without over-approximation). A formal proof of this fact would be rather long and complicated and is beyond the scope of this thesis.

Algorithm 1 exactly describes the steps of the presented method in pseudocode. The application of Formula 5.4 on an example in Listing 2.5 looks as follows:

$$SSA_1 \wedge \neg guard\#ls4\%1 \ \wedge guard\#2\%0 \wedge i\#phi2\%1 = i\#phi2\%0 \tag{5.5}$$

We discuss some facts that should be mentioned, but are less important in the following paragraph.

Loops in the C language can be terminated by unsatisfying the loop condition or by a goto statement or break statement inside the loop body. Ultimately, all the places where a loop can terminate have corresponding conditions and guards in the SSA formula. If we use unwinding, we do not have to add such conditions to the formula for every loop iteration. This is possible, because every loop iteration has its own guard and so if the last loop guard holds, all the loop exit conditions from the previous iterations are unsatisfiable. A loop may be terminated also by an assertion. Assertions are not part of the loop exit conditions in 2LS, but they are handled as a separate program entity. We can describe a check for the satisfiability of every assertion in a program by the formula:

$$\bigwedge_{assertion \in PROGRAM} assertion \tag{5.6}$$

In this way, we include the problem of assertion satisfiability into our non-termination check. To avoid this, we replace assertions with assumptions and thus we check only the paths that will not lead to an assertion violation.

We believe now, that regarding the relationship of *Trans* and SSA_k described in this chapter and the non-termination checking method presented above, it is intuitively clear that formulas 5.4 and 4.8 are equisatisfiable. Therefore, our approach can be used to verify non-termination. Referring to what was stated in Section 5.1, we underline that these formulas are equisatisfiable but not equivalent.

Algorithm 1 Singleton Recurrence Set Search

Input: acyclic *SSA* form of the program, limit N for unwinding

Output: non-terminating *error trace* or *empty*

Method:

```
1: for all assertion in $SSA$ do
2:     replace assertion with assumtion
3: end for
4: $k := 1$
5: while true do
6:     let $SSA_k$ be an $SSA$ form unwound $k$ times
7:     let $\mathcal{L}$ be a set of all loops in the $SSA_k$
8:     $\psi := True$
9:     for all $loop \in \mathcal{L}$ do
10:        $\psi := \psi \wedge \neg guardls_{loop}$
11:    end for
12:    $\varphi := False$
13:    for all $loop \in \mathcal{L}$ do
14:        $i := 1$
15:        while $i \leq k$ do
16:            $\chi := loop\_guard\%0$
17:            for all $phi\_var\%i \in loop$ do
18:                $\chi := \chi \wedge phi\_var\%i = phi\_var\%0$
19:            end for
20:            $\varphi := \varphi \vee (\chi)$
21:            $i := i + 1$
22:        end while
23:    end for
24:    solve $SSA_k \wedge \psi \wedge \varphi$
25:    if SAT then
26:        return error trace
27:    else
28:        if $k \leq N$ then
29:            $k := k + 1$
30:        else
31:            return empty
32:        end if
33:    end if
34: end while
```

Chapter 6

Periodical Recurrence Set Search Algorithm

In this chapter we introduce the derivative algorithm of a known concept [11] in order to cover the small class of programs that require too many loop unwindings to be able to prove the existence of a recurrence set using the method in Chapter 5. The main task of the method introduced in [11] is to prevent a bounded model checker from enumerating a large number of spurious counterexamples, while traversing a loop body. While authors in [11] use under-approximation of a loop in the form of auxiliary paths, we use an over-approximation of loop paths, because we do not consider loop exit conditions. However, they use the same technique, using solution of recurrence relations describing the effect of arbitrarily many loop iterations. In this work we use it to accelerate non-termination check. We designed a method which allows us to reduce a very large number of loop unwindings and solver calls into just one loop unwinding and a few solver calls, for some programs. Our analysis is restricted to study the loops that change the values of variables in every loop iteration according to the pattern of the recurrence relation defined below:

$$x_n = x_{n-1} + c \tag{6.1}$$

Note that c is a constant here. The solution of the recurrence relation above has a form:

$$x_n = x_0 + c \cdot n \tag{6.2}$$

If we are able to prove that a loop defines a recurrence relation of the specified form, we attempt to accelerate Algorithm 1.

We provide a graphical representation of the new method in Figure 6.1. It is the simplified graphical representation of the following program:

```
unsigned int i = nondet_int() % 2;
while (i != 5)
  i += 2;
```

Let us for simplicity restrict the size of the integer to 6. The circles on the figure marked with **S** are start states, where $i = 0$ or $i = 1$. The constant **C** equals 2, according to our program. The addition operation on unsigned integer of the size 6 in the C language has the same behavior as the addition in modular arithmetic in the set $\mathbb{Z}_6$. Our program has two candidates for a periodical recurrence set. One is the set $\{0, 2, 4\}$ and the second is the

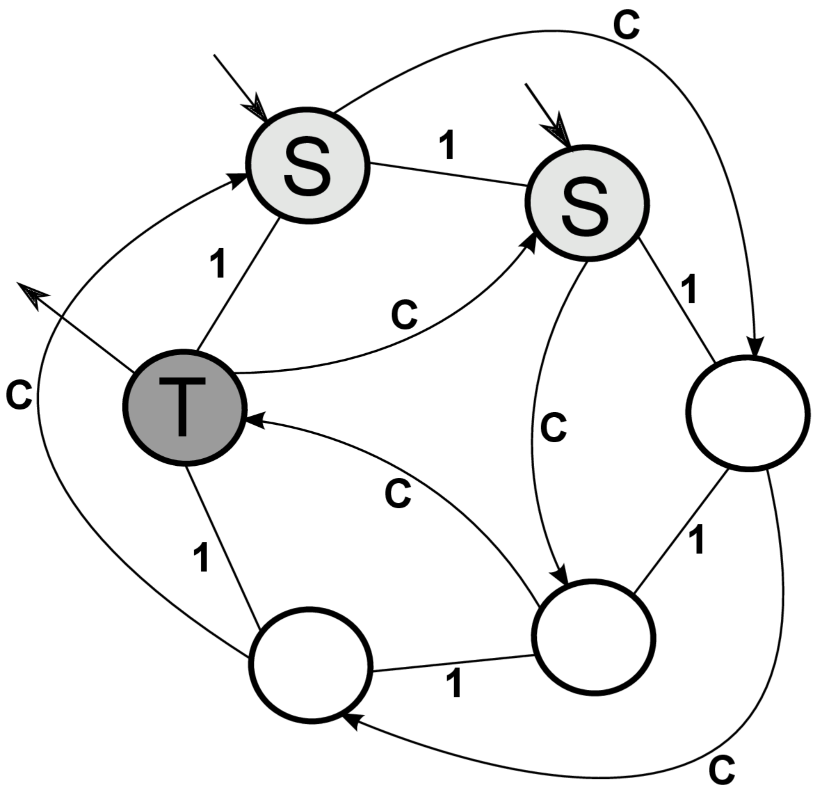

Figure 6.1: CFG with the corresponding parts of the SSA form

set $\{1,3,5\}$. We will prove that to check every such candidate takes one solver call in our method and we further present this method in detail.

We introduce a definition of periodic recurrence set whose existence is a non-termination proof in this method.

Definition 6.0.1. (Periodical Recurrence Set) A transition relation Trans_{loop} with initial states *Init* has a *periodical recurrence set* of states *RSet* iff Formulas 6.3 hold.

$$\begin{aligned}&\exists \mathbf{x}, \mathbf{x}' : \mathit{Trans}_{stem}(\mathbf{x}, \mathbf{x}') \wedge \mathit{RSet}(\mathbf{x}') \\ &\forall \mathbf{x} \exists \mathbf{x}' \exists ! \mathbf{C} : \mathit{RSet}(\mathbf{x}) \implies \mathit{Trans}_{loop}(\mathbf{x}, \mathbf{x}') \wedge \mathit{RSet}(\mathbf{x}') \wedge \mathbf{x}' = \mathbf{x} + \mathbf{C}\end{aligned} \tag{6.3}$$

6.1 Detection of Linearly Changed Phi Nodes in a Loop

In this section we use simplified abstract representation of a loop and program states in order to plainly present the principle of our approach. Let us consider every loop body being a function $f(\mathbf{x}) : \mathbb{Z}_k^n \to \mathbb{Z}_k^n$ over the program variables that takes a vector of bit-vectors and returns a vector of the same type or has an undefined result. The vector only contains the program variables. In terms of this analysis we are interested in loops with $f(\mathbf{x})$ constrained by the formula:

$$\forall \mathbf{x} \exists ! \mathbf{c} : f(\mathbf{x}) \neq \mathrm{UNDEF} \implies f(\mathbf{x}) = \mathbf{x} + \mathbf{c} \tag{6.4}$$

For cases where $f(\mathbf{x})$ is undefined a program cannot reach the end of a loop body. This can be caused by a termination from inside a loop body or by non-termination within a loop body.

Because the SAT solver is unable to deal with a quantifier $\exists!$ (there exists one and only one), we check the property in Formula 6.4 in two steps. At first we check the formula below:

$$\exists \mathbf{x}, \mathbf{c} : f(\mathbf{x}) = \mathbf{x} + \mathbf{c} \tag{6.5}$$

If Formula 6.5 is satisfiable, it gives us a model $\mathcal{M} = \{(\mathbf{x}, \mathbf{X}), (\mathbf{c}, \mathbf{C})\}$. We use it for the second check:

$$\exists \mathbf{x}, \mathbf{c} : f(\mathbf{x}) = \mathbf{x} + \mathbf{c} \wedge \mathbf{c} \neq \mathbf{C} \tag{6.6}$$

The constant vector $\mathbf{C}$ in formula 6.6 is taken from the model in Formula 6.5. The unsatisfiable result of the previous formula means that either $\mathbf{C}$ is the only constant vector to satisfy the formula or $f(\mathbf{x})$ is undefined. As will be shown later we can specify all the cases when the result of $f(\mathbf{x})$ is undefined and thus the previous two checks are sufficient to detect a loop which linearly changes its phi nodes and can be analysed.

6.2 Checking Existence of a Periodical Recurrence Set in a Loop

In this section, we introduce the abstract principle of the non-termination check designed in our method and we present a formal proof that our template covers the whole, potentially non-terminating program path, regarding bit-vectors. We assume that this method receives an input which passed the first check for linearity. We use simplified abstract representation in this section, as was used in Chapters 3 and 4. Note that the vectors here represent the program states and they contain the program counter as well. We use a special notation $\langle \mathbf{k} \rangle$ for a vector, where all elements have the same value k. The constant vector $\mathbf{C_{pc}}$ is created from the vector $\mathbf{C}$ in the previous section by resizing with 0 in place of the program counter, which means that we get back to the loop head and potentially never stop looping. The formula to check non-termination of a loop looks as follows:

$$\exists \mathbf{x}, \mathbf{x}' \forall \langle \mathbf{k} \rangle : \mathit{Trans}_{\mathit{stem}}(\mathbf{x}, \mathbf{x}') \wedge \mathbf{x}'' = \mathbf{x}' + \mathbf{C_{pc}} \cdot \langle \mathbf{k} \rangle \wedge \mathit{Trans}_{\mathit{loop}}(\mathbf{x}'', \mathbf{x}'' + \mathbf{C_{pc}}) \tag{6.7}$$

The initial loop input $\mathbf{x}'$ was generalized for every loop iteration as $\mathbf{x}''$ and it was restricted by the invariant that says the original value of $\mathbf{x}'$ can be increased only by the Hadamard product $\mathbf{C} \cdot \mathbf{k}$. This invariant represents the base of the method. We utilize $\mathit{Trans}_{\mathit{loop}}$ to express that from every state $\mathbf{x}''$, along the non-terminating program path, we can reach the loop head again.

We show here, that if we work with bit-vectors, $\mathbf{x}''$ represents every state reachable from $\mathbf{x}'$, i.e. verifying the values specified by the invariant is the same as verifying every state along a non-terminating program path. In this way we also show the completeness of our method. We know that unsigned integers cannot overflow from the C99 standard[1] §6.2.5/9:

A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

The standard says that the definitions of addition and multiplication on unsigned integers creates a well known ring $(\mathbb{Z}_n, +, 0, -, \cdot, 1)$.

[1] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

We want to prove that our formula covers all the states along a potentially non-terminating program path. For simplicity we use a single bit-vector to prove a property that trivially holds for a vector of bit-vectors. The paths we study may be terminating for some state and in that case the relation Trans_{loop} in formula 6.7 does not hold for some $\mathbf{x''}$. If it holds for every state, we need to prove there is no state left that has not been checked. The following equation is defined for the ring $(\mathbb{Z}_{\text{UINT\_MAX}+1}, +, 0, -, \cdot, 1)$ and it says that the range of the unsigned integer is sufficient to cover every non-terminating path of a loop where all variables are changed by some constant C in every iteration.

$$x + C \cdot \text{UINT\_MAX} = x - C \tag{6.8}$$

The formal proof of Equation 6.8 for unsigned bit-vectors looks as follows:

$$\begin{array}{rcll} x + C \cdot \text{UINT\_MAX} & = & x - C & / + (-x) \\ C \cdot \text{UINT\_MAX} & = & -C & / + C \\ C \cdot \text{UINT\_MAX} + C & = & 0 & /\mathit{distributivity} \\ C \cdot (\text{UINT\_MAX} + 1) & = & 0 & \\ C \cdot 0 & = & 0 & \end{array} \tag{6.9}$$

Returning to Formula 6.7, we face the major problem of methods checking non-termination which is quantifier alternation. We have to avoid the use of the quantifier $\forall$ and replace it with $\exists$. In our case $\forall\langle\mathbf{k}\rangle$ describes every state on the program path. If we use $\exists\langle\mathbf{k}\rangle$ here, we describe only one state that is chosen from the whole path. In this way we can change our search for a non-terminating path to a search for terminating paths, where the sufficient condition is an existing state $\mathbf{x''}$ for which the predicate $\mathit{Trans}_{loop}(\mathbf{x''}, \mathbf{x''} + \mathbf{C_{pc}})$ does not hold. By alternating the quantifier in the formula, we can iteratively enumerate all terminating paths and build a constraint. If we are not able to find a new terminating path, we simply check that a path still exists and if so, it must be non-terminating. Thus we split the check into two steps. In the first step we build a constraint that will exclude all terminating paths by iteratively evaluating the following formula:

$$\begin{aligned} \exists \mathbf{x}, \mathbf{x'} \exists \langle\mathbf{k}\rangle : \quad & \mathit{Trans}_{stem}(\mathbf{x}, \mathbf{x'}) \wedge \mathbf{x''} = \mathbf{x'} + \mathbf{C_{pc}} \cdot \langle\mathbf{k}\rangle \wedge \neg \mathit{Trans}_{loop}(\mathbf{x''}, \mathbf{x''} + \mathbf{C_{pc}}) \wedge \\ & \mathit{constraints}(\mathbf{x}) \end{aligned} \tag{6.10}$$

We update the predicate $\mathit{constraints}(\mathbf{x})$ in every iteration using the model we get from Formula 6.10. Let $\mathbf{X''}$ be a model of $\mathbf{x''}$. We want to add some constraint to the formula so that in the next iteration $\mathbf{x''}$ will get a different value. The constraint could have a simple form $\mathbf{x''} \neq \mathbf{X''}$. However, we know that if a loop potentially reaches the state, where $\mathbf{x''} = \mathbf{X''}$, then all the states that lead to that given state and also all the states which will follow from that state can be excluded. The invariant $\mathbf{x''} = \mathbf{x'} + \mathbf{C_{pc}} \cdot \langle\mathbf{k}\rangle$ is used to describe all such states. We may use a larger restriction that will exclude every value of $\mathbf{x''}$ such that $\mathbf{x''} = \mathbf{X''} + \mathbf{C_{pc}} \cdot \langle\mathbf{k}\rangle$. Since $\mathbf{x''}$ is computed from $\mathbf{x'}$ in the same way, we can interchange a value of $\mathbf{x'}$ and $\mathbf{x''}$. To exclude all specified values would lead to quantifier alternation again. Therefore we do a compromise and find a property that describes as many such values as possible. Such a property could be componentwise modulo, since $(\mathbf{X'} + \mathbf{C_{pc}} \cdot \langle\mathbf{k}\rangle)\%\mathbf{C_{pc}}$ always equals $\mathbf{X'}\%\mathbf{C_{pc}}$ if it does not exceed the size of the bit-vector. Thus, the constraint added in every step has the form $\mathbf{x'}\%\mathbf{C_{pc}} \neq \mathbf{X'}\%\mathbf{C_{pc}}$, where $\mathbf{X'}$ is taken from a model of the formula as a value of $\mathbf{x'}$. Note that the cases where we use modulo 0 are defined as $n\%0 = n$.

We update the constraints until the formula becomes unsatisfiable. Then the predicate $constraints(\mathbf{x})$ looks as follows:

$$\bigwedge_{\mathbf{x}' \in \text{MODELS of THE FORMULA } 6.10} \mathbf{x}' \% \mathbf{C_{pc}} \neq \mathbf{X}' \% \mathbf{C_{pc}} \tag{6.11}$$

After finishing the first step we are able to exclude every terminating path in the loop. The second step only verifies that there is still some program path reaching the loop left and it is indeed non-terminating. The formula for the second step is as follows:

$$\exists \mathbf{x}, \mathbf{x}' \exists \langle \mathbf{k} \rangle : Trans_{stem}(\mathbf{x}, \mathbf{x}') \wedge \mathbf{x}'' = \mathbf{x}' + \mathbf{C_{pc}} \cdot \langle \mathbf{k} \rangle \wedge constraints(\mathbf{x}) \tag{6.12}$$

6.3 Generalization of the Periodical Recurrence Set Search

In the previous two sections we used the vector $\mathbf{x}$ as a parameter for the function $f(\mathbf{x})$. Instead of $\mathbf{x}$ which represents all the variables in the program, we use only phi nodes of a specific loop. We present a generalization of our method in order to better explain our approach. Let us take the program:

```
unsigned c, i = 1;
while (1)
{
  if (c)
    i = i + 1;
  else
    i = i + i;
}
```

As we can see the variable c does not belong to the set of phi nodes of the given loop. Nevertheless the form of function $f(\mathbf{x})$ depends on its value. We split vector $\mathbf{x}$ into two vectors $\mathbf{x}'$ and $\mathbf{x}''$, where vector $\mathbf{x}'$ represents the variables changed inside a loop and $\mathbf{x}''$ describes the rest of variables in a program. The generalization of Formula 6.4 has the following form:

$$\exists \mathbf{x}'' \forall \mathbf{x}' \exists ! \mathbf{c} : g(\mathbf{x}'') = f(\mathbf{x}') = \mathbf{x}' + \mathbf{c} \tag{6.13}$$

Considering the program example above, the function $g(\mathbf{x}'')$ is defined as follows:

$$g(\mathbf{x}'') = \begin{cases} \mathbf{x}' + (1) & \mathbf{x}'' \neq (0) \\ \mathbf{x}' + \mathbf{x}' & \mathbf{x}'' = (0) \end{cases} \tag{6.14}$$

As we can see formula 6.13 is satisfiable for $\mathbf{x}'' \neq (0)$. The problem of the general approach is quantifier alternation. We restricted ourselves to only use its simplified version:

$$\forall \mathbf{x}'' \forall \mathbf{x}' \exists ! \mathbf{c} : g(\mathbf{x}'') = f(\mathbf{x}') = \mathbf{x}' + \mathbf{c} \equiv \forall \mathbf{x} \exists ! \mathbf{c} : f(\mathbf{x}) = \mathbf{x} + \mathbf{c} \tag{6.15}$$

6.4 Implementation of Periodical Recurrence Set Search Algorithm

This section describes the presented algorithm in a form that is implemented in 2LS. We use the representation already described in Section 5.1. The formulas have the form fed to the solver, and we also deal with implementation details, which we were unable to discuss in the context of previous sections, because we did not use the SSA-based representation. Algorithm 2 represents our implementation of the presented algorithm in pseudocode.

6.4.1 Implementation of Detection of Linearly Changed Phi Nodes

Formula 6.10 is passed to the solver in the form:

$$SSA_k \wedge \bigwedge_{phi\_var \in loop} phi\_var\%k + c = phi\_var\%(k-1) \tag{6.16}$$

We verify this formula for every loop separately. SSA_k defines a function $f(\mathbf{x})$ (see Section 6.1) for every loop body. Constant c is unique for every phi_var. We use the over-approximation to verify the formula, because we have to check all values for every phi node. Recall that the value of $phi\_var\%k$ in SSA is specified by the equation $phi\_var\%k == guardls ? var\_lb : var\_x$, where var_lb is a free variable (see Section 2.3). Therefore, the solver can assign every possible value to the variable $phi\_var\%k$. The value of the variable $phi\_var\%(k-1)$ is derived from the equations of a loop body and the previous value of a phi node represented by $phi\_var\%k$. We provide an example of the application of formula 6.16 on the example in Listing 2.5:

$$SSA_1 \wedge i\#phi2\%1 + c = i\#phi2\%0 \tag{6.17}$$

Formula 6.6 is passed to the solver in the form:

$$SSA_k \wedge \bigwedge_{phi\_var \in loop} phi\_var\%k + c = phi\_var\%(k-1) \wedge c \neq C \tag{6.18}$$

We take model C of the constant c if Formula 6.16 is satisfiable and we use it for the second check. Below we provide an example of the application of formula 6.18 on the example in Listing 2.5.

$$SSA_1 \wedge i\#phi2\%1 + c = i\#phi2\%0 \wedge c \neq C \tag{6.19}$$

6.4.2 Implementation of The Non-termination Check

In this section we describe the non-termination check of the method whose theoretical principles are listed in Section 6.2. Recall, that we check non-termination only if the candidate loop passes the check in the previous section.

At first we show how invariant $\mathbf{x}'' = \mathbf{x}' + \mathbf{C_{pc}} \cdot \langle \mathbf{k} \rangle$ is represented in the SSA form. The loop head equations $phi\_var\%k == guardls\,?\,var\_lb : var\_x$, where var_lb is a free variable, are supplemented by constraints. These constraints have the form of an invariant which is an equation $var\_lb == var\_x + C \cdot k$. The character C denotes a constant computed by the linearity check and k is a free variable. Remember that k is the same variable for every phi node in the loop, but C is computed for every phi node separately. We utilize over-approximation which gives us the ability to check every state along the program path at once without iterative loop unwinding. The *guardls* for the currently analyzed loop must hold in order to apply the constraints. As we iteratively check every loop, we need to avoid the over-approximation of the other loops in order to preserve completeness. This approach is already described in Section 5.2 in Formula 5.2, but we omit *guardls* for the current loop.

The relations $Trans_{stem}$ and $Trans_{loop}$ are encoded in the SSA form as described in Section 5.1. In Section 6.1, we mention that we are able to detect undefined behavior of the function $f(\mathbf{x})$. We defined a termination condition or rather the condition for which a given loop path does not fulfill non-termination requirements in the form $\neg Trans_{loop}(\mathbf{x}'', \mathbf{x}'' + \mathbf{C_{pc}})$, where $\mathbf{x}''$ represents any state along the path. Let us first analyze when a loop that passed the linearity check can have an undefined behavior. Note that neither formula 6.16 nor formula 6.18 includes a loop guard. Therefore, the only cause for undefined behavior is that a loop guard of the subsequent loop unwinding does not hold. In other words, for every state of a non-terminating path the loop guard of the subsequent loop unwinding must hold. The formula for the first step of the non-termination check looks as follows:

$$\exists \ell \exists var\_x : SSA_k \wedge \bigwedge_{loop \in LOOPS} loop \neq analysed\_loop \implies \neg guard\_ls_{loop} \wedge \\ \bigwedge_{phi\_var\%k \in loop\%k} var\_lb == var\_x + C \cdot \ell \wedge constraints(\mathbf{x}) \wedge \neg loop\_guard\%(k-1) \tag{6.20}$$

We provide an example of the application of the formula above for listing 2.5. The formula of the first iteration for empty constraints looks as follows:

$$SSA_1 \wedge i\#lb4\%1 = i\#1 + 1.\ell \wedge \neg guard\#2\%0 \tag{6.21}$$

No $\neg guardls$ is used, because the program has only one loop. In the first iteration, the solver provides us with a model. One of the valid models for our formula is $\mathcal{M} = \{(i\#lb4\%1, 0), (i\#1, 1), (l, 1), ...\}$. We use it to create a constraint in the form $i\#1\%C \neq 1\%1$. We create a new formula with an additional constraint:

$$SSA_1 \wedge i\#lb4\%1 = i\#1 + 1.\ell \wedge i\#1\%1 \neq 1\%1 \wedge \neg guard\#2\%0 \tag{6.22}$$

The formula above is no more satisfiable. And the algorithm can proceed to the second check.

At the end we have to check whether there still exists some path that satisfies the constraints created in the first step. The formula is the same as for the first step except the loop guard at the end. We do not have to check the satisfiability of the loop guard, since every path that violates it has been already excluded. The check looks as follows:

$$\exists \ell \exists var\_x : SSA_k \wedge \bigwedge_{loop \in LOOPS} loop \neq analysed\_loop \implies \neg guard\_ls_{loop} \wedge \\ \bigwedge_{phi\_var\%k \in loop\%k} var\_lb == var\_x + C \cdot \ell \wedge constraints(\mathbf{x}) \tag{6.23}$$

In reference to our practical example, we now check the formula:

$$SSA_1 \wedge i\#lb4\%1 = i\#1 + 1.\ell \wedge i\#1\%1 \neq 1\%1 \tag{6.24}$$

It is unsatisfiable, because created constraints excluded every path through the loop. Therefore, for this example, the result of the method is DON'T KNOW.

Algorithm 2 Periodic Recurrence Set Search

Input: acyclic SSA_k form of the program with replaced assertions unwound k times
Output: non-terminating *error trace* or *empty*
Method:

```
1:  let $\mathcal{L}$ be a set of all loops in the $SSA_k$
2:  for all $loop \in \mathcal{L}$ do
3:      $\varphi := SSA_k$
4:      for all $phi\_var\%k \in loop\_body\%k$ do
5:          $\varphi := \varphi \wedge phi\_var\%k + const_{phi\_var} = phi\_var\%(k-1)$
6:      end for
7:      solve $\varphi$
8:      if UNSAT then
9:          continue
10:     end if
11:     let $\Phi$ be a model of the formula $\varphi$
12:     for all $phi\_var\%k \in loop\_body\%k$ do
13:         $\varphi := \varphi \wedge const_{phi\_var} \neq \Phi(const_{phi\_var})$
14:     end for
15:     solve $\varphi$
16:     if SAT then
17:         continue
18:     end if
19:     $\psi := SSA_k$
20:     for all $loop_2 \in \mathcal{L}$ do
21:         if $loop_2 \neq loop$ then
22:             $\psi := \psi \wedge \neg guardls_{loop_2}$
23:         end if
24:     end for
25:     for all $phi\_var\%k \in loop\_body\%k$ do
26:         let $phi\_var\%k = guardls\ ?\ loop\_back\_var\ :\ above\_var$
27:         $\psi := \psi \wedge loop\_back\_var\%k = above\_var + l.\Phi(const_{phi\_var})$
28:     end for
29:     while solve $(\psi \wedge \neg loop\_guard\%(k-1))$ = SAT do
30:         let $\Psi$ be a model of the formula $\psi$
31:         $\chi = False$
32:         for all $phi\_var\%k \in loop\_body\%k$ do
33:             let $phi\_var\%k = guardls\ ?\ loop\_back\_var\ :\ above\_var$
34:             $\chi := \chi \vee above\_var \neq \Psi(above\_var)$
35:         end for
36:         $\psi := \psi \wedge (\chi)$
37:     end while
38:     if solve $\psi$ = SAT then
39:         return error trace
40:     end if
41: end for
42: return empty
```

Chapter 7

Implementation

We implemented our analysis in a separate module called **summary_checker_nonterm**. Non-termination analysis is in 2LS, available under the option **--nontermination**. It automatically uses function inlining (option **--inline**), as our method is intraprocedural. We also use automatic substitution of assertions with assumptions to exclude all paths where any assertion can be violated. Function inlining allows us to use one solver instance. As for the solver, we use default option in 2LS which is MiniSat 2.2.1. We update the solver content incrementally in every iteration with new unwindings. Once the program is initially transformed into SSA form, this ensures efficient run of the analysis.

The formulas added to the solver always have a form of implication $enabling\_expr \implies added\_formula$. Loop unwinding is not monotonic, which means that some formulas are added to the solver, but there are some that need to be removed to preserve correctness. We remove the formulas by adding the negations of their enabling expressions into the solver. In this way we can incrementally update the SSA formula without the need to pop the solver stack, which is not supported by SAT solvers. However, conjunction of enabling expressions and formula of a singleton recurrence set are added to the solver in a separate context which is newly created in every iteration. As we can see a part of Formula 5.4 is a big disjunction updated in every iteration of the algorithm. The solver implementation in 2LS does not allow us to update subformulas in the current formula. Therefore we always create new context where we put these temporary formulas in.

The module **summary_checker_nonterm** containing the implementation of the non-termination analysis was conformed to the uniform template used by abstract interpretation - module **summary_checker_ai**, bounded model checking - module **summary_checker_bmc** and kIkI - module **summary_checker_kind**. The termination analysis is included in the module of the abstract interpretation. All these analyses inherit from the base class **summary_checker_base** that provides these methods with the common functionality. Every method has its own unwinding concept and additional formulas. Bounded model checking unwinds loops iteratively whilst abstract interpretation does it only once. Nevertheless, all the analyses call a method **check_properties** at some point, which is part of the base class and no analysis overrides it except the non-termination checker.

The method **check_properties** verifies if some assertions in a program have been violated. In the case of non-termination analysis, we need to check if there exists a recurrence set in the current unwinding of the SSA form. Therefore, the properties we check are not assertions but loops. We keep this adjusted common template to avoid big changes in the code and use existing algorithms as much as possible. We utilize the original concept of

property_map which is C++ container `std::map` devoted to gather studied properties, which are loops in our case. It creates an interface between callee and caller. The result of every property is by default set to *unknown*. If non-termination is detected by the solver it is changed to *fail* and this information is important for the generation of error trace. The generation of error traces is already implemented in 2LS. We wanted to use this implementation with minimal changes. A check of all properties is implemented in the class **cover_goals_extt**. It calls the solver, checks spurious counterexamples (this does not have to be done for non-termination analysis) and builds an error trace when an error is found.

We implemented two separate methods for non-termination analysis, but in the end, we integrate them into one. After a certain number of unwindings performed by the singleton recurrence set method, there is one step devoted to employing the periodical recurrence set method. If it does not succeed we continue unwinding with the first method. We did not implement any heuristics for the second method which would give a reason to elaborate the mentioned integration. For example the use of a heuristic that detects that the loop is fully unwound and forbids over-approximation for such a loop would give a reason to apply the second method not only once, but after every x unwindings and also if a loop is fully unwound.

Chapter 8

Experiments

We benchmarked our implementation with *BenchExec 1.9 framework*[1] [2]. This framework has been in use on International Competition on Software Verification[2] since 2012. We used the set of benchmarks[3] from SV-COMP 2017 [1]. The BenchExec allows us to reliably measure and limit resources as wall time, CPU time and memory usage. We tested our method exclusively in category Termination and we compared our results with those from SV-COMP 2017[4]. The category Termination has currently 1437 benchmarks from which 940 are classified as terminating and 497 as non-terminating. We also supported the faster benchmarking by our own script that is measuring the number of unwindings used to prove non-termination and tests non-terminating and terminating benchmarks separately.

2LS and CBMC have bash wrappers[5] that provide an interface for BenchExec. We exploit the wrapper for 2LS to integrate termination and non-termination analysis into the one method. The original implementation of termination analysis in 2LS was able to prove never-termination, if it was not able to find a path from the beginning of the function to its end [5]. However, this approach has some existing flaws that caused incorrect detections of non-termination. We have prevented the termination analyzer from checking never-termination and we completely rely on the non-termination analyzer, since it covers all benchmarks where termination analyzer detects non-termination. The analyzers are run in parallel in the background subshell.

We used the same resource limits as in SV-COMP 2017. 8 processing units, 15GB memory limit and 15 minutes of CPU time for each verification run. The limit for witness validation was 2 processing units, 7GB of memory and 1.5 minutes of CPU time for violation witnesses and 15 minutes of CPU time for correctness witnesses. All of the measurements were made on the machine with parameters CPU: Intel Core i7-6700 CPU 3.40GHz, cores: 8, frequency: 4000 MHz with Turbo Boost enabled, RAM: 33618 MB. Our operating system was Linux x86_64 with Ubuntu-16.04 and Linux kernel 4.4.

We applied the same schema to compute the score that was used for SV-COMP 2017 [1]. If a tool reports the correctness of a correct program and if a validator does confirm the witness, the score is 2, otherwise it is 1. If the correctness is proved for an incorrect program the score is -32. The witnesses for property violations are not checked. If the

[1] https://github.com/sosy-lab/benchexec
[2] https://sv-comp.sosy-lab.org/2017/
[3] https://github.com/sosy-lab/sv-benchmarks/tree/master/c
[4] https://sv-comp.sosy-lab.org/2017/results/results-verified/META_Termination.table.html
[5] https://github.com/diffblue/cprover-sv-comp

property violation is correctly found the score is 1, otherwise, in the case of false alarm it is −16. Any error or unknown result does not influence the final score.

The benchmarking was done in order to see the improvement of 2LS on SV-COMP 2017 benchmark set in the category *Termination*. Our aim was also to find existing bugs in our implementation.

In the following section, we present the results achieved by 2LS with the implementation of singleton recurrence set (SRS) method for non-termination analysis. We also mention the principles of the simple parallel procedure (take the first result) that was used in the wrapper script to benchmark the tool with BenchExec. The results of the version, where the periodical recurrence set (PRS) method was supplemented to the analysis, are presented in Section 8.2. In that section, we also present the improvement for our parallel procedure (take the first valid result). We compare our results with the competitive tools in Section 8.3. In the last section, we give a short bug report that describes flaws found in the current version of 2LS.

8.1 Benchmarking of the Singleton Recurrence Set Method

In Table 8.1 we compare the results of the singleton recurrence set method with the results of the version competing on SV-COMP 2017. The unifying procedure of our two algorithms for this table was to take the first available result of the analysis, even when it was *unknown*. We see that the number of detected *correct true* benchmarks decreased, while the number of correctly detected false properties significantly increased, but the most important difference is the reduction of incorrect results from 34 to 6. We run the two processes in parallel and they share all the resources. Hence, in cases where proving termination with the previous implementation took significant amount of time or memory, we may reach the limit of resources before the result can be computed. Recall, that non-termination analysis is doing unwinding which is a greedy process for a memory. If tested exclusively, the number of benchmarks where our method is able to detect non-termination is slightly greater (475) than we present in Table 8.1 (465). The reason of this is that in some cases termination analysis returns an *unknown* result before non-termination is proved. Generally to disprove the property takes less time and as we see non-termination was detected faster than incorrect termination. The reduction of incorrect false results is mainly caused by delegating non-termination analysis exclusively to the non-termination analyzer as we mentioned at the beginning of this chapter. In some cases it was caused by the *unknown* result of the termination analyzer that finished sooner. In the end we see that the major advantage of this parallel approach is a reduction of the number of incorrect results.

The score we give in Table 8.1 is not normalized [1]. We see how important it is to avoid incorrect results if we look at the penalty for incorrectness of the tool. The improvement made by non-termination analysis would not be so significant without hiding so many errors using our parallel approach. The score achieved for *correct false* detections was improved by 34% and the total score by 65.6%, which is very promising.

The amount of resources used has no effect on the score, but it plays a role when it comes to effectiveness of the method. Probably the most important aspect is time. We were able to reduce total time consumption more than three times. As we can see in Table 8.2 our method used 7530 of 17400 seconds to prove correct results, whilst the original method used only 5660 of 60800. The method is able to resolve many of the benchmarks that were time-consuming or even led to timeout. On the other hand, we see that the time of termination analysis is worse. Recall that we use parallelism and therefore the CPU

	2LS 0.5.0 - SV-COMP 2017		2LS 0.5.1 - SRS Method	
	Status	Score	Status	Score
total	1437	899	1437	1489
correct results	927	1507	1025	1585
correct true	580	1160	560	1120
correct false	347	347	465	465
incorrect results	34	-608	6	-96
incorrect true	4	-128	0	-
incorrect false	30	-480	6	-96

Table 8.1: Score - comparison with the results from SV-COMP 2017

time (amount of time that a task spent on different CPUs) consumption is doubled whilst both methods are running at the same time. The amount of time taken by incorrect results increased even though the number of incorrectly classified benchmarks was significantly decreased. One of the benchmarks took 120 seconds to prove non-termination. The source file had 75.8KB size with 3694 lines of code. We analyzed the error trace and it was a correct non-terminating program trace. We checked the competition results and there was no tool able to prove either termination or non-termination of this incorrectly classified benchmark. We found six other incorrectly classified benchmarks, which are reported and discussed later.

Memory consumption increased with the use of our method. The non-termination analysis iteratively unwinds a program and runs the solver. This is a memory consuming process, especially when it comes to nested loops, where the number of loop unwindings grows exponentially with the depth. Both analyses run in parallel and thus the amount of used memory is considerable.

	2LS 0.5.0 - SV-COMP 2017		2LS 0.5.1 - SRS Method	
	CPU Time (s)	Memory (GB)	CPU Time (s)	Memory (GB)
total	60800	447	17400	1320
correct results	5660	268	7530	774
correct true	5480	253	6980	709
correct false	181	14.2	544	64.7
incorrect results	103	17.0	122	15.0
incorrect true	1.23	.109	-	-
incorrect false	102	16.9	122	15.0

Table 8.2: Resources - comparison with the results from SV-COMP 2017

We further studied the number of unwindings needed for specific benchmarks and we tested the non-terminating and terminating sets of benchmarks separately in order to discover as many flaws in the implementation as possible.

The results of the method searching for the singleton recurrence set brought the following findings:

- We found 11 benchmarks in the sv-benchmarks test set with an overflow issue that were either fixed or moved to the todo file group - pull request to sosy-lab/sv-benchmarks[6] by Peter Schrammel[7] (these benchmarks were either detected as non-terminating or our program finished on limit).

- 7 benchmarks were proven to be non-terminating and incorrectly classified.
- 5 benchmarks were incorrectly proven to be non-terminating which pointed to two existing bugs in 2LS.
- Non-termination analysis did not finish on 10 benchmarks in the given amount of time.
- 1 specially difficult benchmark was proved to be incorrectly classified without having an overflow issue.
- Our tool was able to detect every single non-terminating benchmark in the folder **product-lines**. 5 of these benchmarks were not detected by any other tool.

As mentioned above we updated the set of benchmarks according to our findings. Four of them were moved to category overflow, one benchmark was moved from the class of non-terminating to the class of terminating benchmarks and an overflow trace has been removed in seven benchmarks. The overview of the files is shown in Table 8.3.

Benchmarks with the overflow trace removed
termination-crafted/Mysore_false-termination_true-valid-memsafety.c
termination-memory-alloca/PodelskiRybalchenko-2004VMCAI-Ex2-alloca_false-termination.c
*termination-crafted-lit/AliasDarteFeautrierGonnord-SAS2010-loops_true-termination_false-no-overflow.c
*termination-crafted-lit/PodelskiRybalchenko-LICS2004-Fig1_true-termination_false-no-overflow.c
*loops/trex01_false-unreach-call_true-termination.i
*loops/trex01_true-unreach-call_true-termination.i
*termination-15/array17_alloca_true-termination.c.i
Benchmarks removed from the test set because of overflow
termination-crafted/Singapore_plus_false-termination_true-valid-memsafety.c
termination-crafted/Singapore_v1_false-termination_true-valid-memsafety.c
termination-crafted/Singapore_v2_false-termination_true-valid-memsafety.c
*termination-15/array11_alloca_true-termination.c.i
Reclassified benchmarks
*product-lines/elevator_spec13_productSimulator_true-unreach-call_true-termination.cil.c

Table 8.3: Overview of the files where we detected overflow (benchmarks where our method detected non-termination are marked with *)

8.2 Benchmarking of the Periodical Recurrence Set Method

We have also designed and implemented the method that searches for periodic recurrence sets, to cover the maximum number of benchmarks that were not detected by the SRS method. It was precisely four benchmarks, where the SRS method was highly ineffective and the PRS was able to solve it. This did not lead to significant improvements, but we made also other changes discussed in the following two paragraphs.

As mentioned in the previous section we discovered incorrectness in the original set of benchmarks that was thoroughly studied, reported and the pull request to sosy-lab/sv-benchmarks was created. The tests presented in this section were made on the updated set of benchmarks.

[6] https://github.com/sosy-lab/sv-benchmarks
[7] https://github.com/peterschrammel/sv-benchmarks/tree/fix-termination-signed-overflow

We also improved our parallel algorithm so that if the result of termination analysis is *unknown*, the algorithm waits for the result of the second process that is still running. Our method needs one or two unwindings to find termination violation for most of the benchmarks. For three programs it was 50 or more unwindings, where maximum was 211. We decided to set the unwinding limit to 220, which according to our experiments, represents a reasonable trade-off with respect to amount of the used resources.

We present the final score of the combined method, where we integrate PRS and SRS into one analysis, in Table 8.4. It is improved compared to the results in the previous section and the major difference is in the number of *correct false* results. This improvement is caused by updates in the set of benchmarks and the better parallel algorithm. If termination analysis finishes sooner with the result *unknown*, the unifying procedure will still wait for the non-termination analysis to come up with its result. However, because we use the unwinding limit, in two cases it happens that non-termination analysis finishes sooner with the *unknown* result. The unifying procedure ends without waiting for the result of the termination analyzer, hence 558 *correct true* detections.

	2LS 0.5.1 - SRS + PRS Method with Original Parallel Procedure		2LS 0.5.1 - SRS + PRS Method with Updated Parallel Procedure	
	Status	Score	Status	Score
total	1433	1509	1433	1516
correct results	1029	1589	1038	1596
correct true	560	1120	558	1116
correct false	469	469	480	480
incorrect results	5	-80	5	-80
incorrect true	0	-	0	-
incorrect false	5	-80	5	-80

Table 8.4: Score - updated set of benchmarks

Table 8.5 compares the amount of resources used by the SRS + PRS method with the old procedure and by the same method using the updated version of it. Remember that previously we took the *unknown* result of the termination analyzer as the final result of the procedure, but in this method we are waiting for the result of non-termination analysis which is limited by 220 unwindings. This explains both the doubled memory consumption and much worse overall time. However, we were able to fully exploit the potential of our method and increase the score in Table 8.1. We see a trade-off between the amount of resources and the achievable score.

8.3 Comparison with the Other Tools

This section is devoted to comparison with the tools awarded in category *Termination* on SV-COMP 2017. Note that the score we use in this chapter is not normalized, but is directly taken from the results of Benchexec. The tools we have chosen for the comparison are **UAutomizer**, which is the winner in the category *Termination* and **AProVE** that finished second in this category.

We use the results from SV-COMP 2017 achieved by the other tools tested on a different machine:

- CPU: Intel Xeon E3-1230 v5 3.40 GHz

	2LS 0.5.1 - SRS + PRS Method with Original Parallel Procedure		2LS 0.5.1 - SRS + PRS Method with Updated Parallel Procedure	
	CPU Time (s)	Memory (GB)	CPU Time (s)	Memory (GB)
total	19000	1300	86200	2510
correct results	8810	767	8320	743
correct true	8050	688	7520	662
correct false	754	79.8	794	80.7
incorrect results	2.37	.265	2.18	.259
incorrect true	-	-	-	-
incorrect false	2.37	.265	2.18	.259

Table 8.5: Resources - comparison with the results from SV-COMP 2017

- cores: 8
- frequency: 3.8 GHz, Turbo Boost: disabled
- RAM: 33553 MB
- system: Linux 4.4.0-59-generic

Our computer had a slightly better performance rating than the machines used at SV-COMP 2017[8] and we also used Turbo Boost 2.0. However, it is important to mention that on our machine we did not run the tests exclusively, but other processes were running alongside the tests. This might influence the number of cache misses and basically slow down the memory access of our tests. Table 8.6 compares the CPU time that was measured at SV-COMP 2017 to the time measured on our machine. The findings of this measurement indicate that the performance of the machines is comparable, since the difference in used total amount of time is only about 3%.

	2LS 0.5.0 - Intel Xeon E3-1230 v5	2LS 0.5.0 - Intel Core i7-6700
	CPU Time (s)	CPU Time (s)
total	58600	60800
correct results	4630	5660
incorrect results	90.6	103

Table 8.6: Machine performance comparison

Based on the results in Table 8.7, we predict that 2LS could finish the second at SV-COMP 2018. To compete for the winning position, the termination analysis must be improved, which is the part of 2LS where further work needs to be done.

We also roughly compare the performance of the tools in Table 8.8 and Table 8.9. As you can see, the efficiency of 2LS is much higher than the efficiency of other successful tools. The UAutomizer tool had a similar number of *correct false* detections as 2LS had, but needed approximately 11 times more time than 2LS. We can see that 2LS used almost half of the total computation time to produce a correct result, while other tools were much worse.

[8] http://cpuboss.com/cpus/Intel-E3-1230V5-vs-Intel-Core-i7-6700

	2LS 0.5.1 - SRS Method	UAutomizer	AProVE
	Score	Score	Score
total	1489	2085	978
correct results	1585	2085	978
correct true	1120	1626	916
correct false	465	459	62
incorrect results	122	-	-
incorrect true	-	-	-
incorrect false	122	-	-

Table 8.7: Score - comparison with the best two other tools

	2LS 0.5.1 - SRS Method		UAutomizer	
	CPU Time (s)	Memory (GB)	CPU Time (s)	Memory (GB)
total	17400	1320	128000	2070
correct true	6980	709	23800	671
correct false	544	64.7	5940	268

Table 8.8: Resources - comparison with the winner

	2LS 0.5.1 - SRS Method		AProVE	
	CPU Time (s)	Memory (GB)	CPU Time (s)	Memory (GB)
total	17400	1320	598000	7140
correct results	6980	709	11100	709
incorrect results	544	64.7	2050	73.9

Table 8.9: Resources - comparison with the second placed tool

8.4 Bug Report

5 incorrectly detected benchmarks reveal 2 hidden bugs in 2LS. In this section, we give information about the 2 bugs discovered in 2LS. One bug caused incorrect results in 4 of these 5 benchmarks. We studied this in depth, but the fix complexity was beyond the scope of our work. We provide a description of our findings which can be used in future development. We use the following program example to explain the cause:

```
typedef struct node {
  struct node* next;
} node_t;
int main(void)
{
  node_t* head = NULL;
  node_t* curr;
  //allocate singly linked list
  for (int i = 0; i < 2; i++) {
    curr = malloc(sizeof(node_t));
    curr->next = head;
    head = curr;
  }
  //iterate singly linked list
  node_t* curr = head;
  while (curr != NULL) {
    curr = curr->next;
  }
  return 0;
}
```

As we can see, the program noted above is always terminating. It is creates a singly linked list and then it searches through it till it reaches the end, which is NULL. Every memory allocation in 2LS is replaced by *dynamic_object$i*, which is a symbolic name for the contents of allocated memory. The unwinding procedure is supposed to create the new unique dynamic object for every unwinding if such object is created inside the loop. In the current version of 2LS, dynamic objects are not replaced with unwinding. Therefore, in our previous example, the non-termination analysis will detect infinite loop execution of the `while` loop after 2 unwindings. The generated SSA form encodes the same program behavior as if `malloc` function would return the same memory address twice in the `for` loop. It creates a recursive loop in the list.

The second bug was not fully documented. We were able to detect that the SSA form is not correctly generated for the nested loops. There exist infeasible paths where we can prove and disprove assertions with the same SSA formula without over-approximation. The benchmark, where the bug was detected is **loops/while_infinite_loop_4_false-unreach-call_true-termination.i**.

Chapter 9

Conclusion

Our aim in this work was to design and implement non-termination analysis within the 2LS tool. The tool uses the SSA-based representation and works exclusively on bit-vectors. This represented the major challenge in this work. Existing methods for proving non-termination were studied, and we implemented the well-known approach introduced by [10] adjusted for the representation used in our tool. Our results show that even a simple algorithm can be highly effective with the proper, SSA-based, representation. We tested our tool on the set of benchmarks from SV-COMP 2017 and we were able to detect non-termination in 475 of 497 non-terminating benchmarks. The method also revealed 1 incorrectly classified benchmark and 6 benchmarks with an overflow issue.

Even though the implemented method produced very good results, we wanted to cover the set of programs where it was ineffective and took an excessive amount of time to compute the result. Therefore, we introduced the concept of periodical recurrence sets and we implemented our second method. This was able to cover 4 more benchmarks and increase the total number of detected non-terminating benchmarks to 479. We designed a parallel algorithm for 2LS that combined our non-termination analysis with the termination analysis already implemented in our tool. The final score in the category *Termination* has been increased from 899 to 1489, which is a significant improvement.

Further research is needed into termination analysis in 2LS in order to increase the number of successfully proved terminating benchmarks and to remove the existing bugs in the analysis. Additional work needs to be performed to improve the results of the unifying procedure for the termination and the non-termination analysis. Research into recursion support is also needed, since more than 100 benchmarks on SV-COMP are currently recursive. The use of more complex templates for periodical recurrence sets could be a productive field of study. We highly recommend studying the use of this approach for interprocedural analysis, which could increase the scalability of the presented method.

Bibliography

[1] Beyer, D.: Software Verification with Validation of Results. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2017. pp. 331–349.

[2] Beyer, D.; Löwe, S.; Wendler, P.: Benchmarking and resource measurement. In *Model Checking Software*. Springer. 2015. pp. 160–178.

[3] Brain, M.; Joshi, S.; Kroening, D.; et al.: Safety Verification and Refutation by k-Invariants and k-Induction. In *International On Static Analysis*. Springer. 2015. pp. 145–161.

[4] Chen, H.-Y.; Cook, B.; Fuhs, C.; et al.: Proving nontermination via safety. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014. pp. 156–171.

[5] Chen, H.-Y.; David, C.; Kroening, D.; et al.: Synthesising Interprocedural Bit-Precise Termination Proofs (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE. 2015. pp. 53–64.

[6] Clarke, E.; Kroening, D.; Lerda, F.: A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2004. pp. 168–176.

[7] Clarke, E.; Kroening, D.; Yorav, K.: Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. 2003.

[8] Cook, B.; Kroening, D.; Rümmer, P.; et al.: Ranking function synthesis for bit-vector relations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2010. pp. 236–250.

[9] Cousot, P.; Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977. pp. 238–252.

[10] Gupta, A.; Henzinger, T. A.; Majumdar, R.; et al.: Proving non-termination. *ACM Sigplan Notices*. vol. 43, no. 1. 2008: pp. 147–158.

[11] Kroening, D.; Lewis, M.; Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. *Formal methods in system design*. vol. 47, no. 1. 2015: pp. 75–92.

[12] Leike, J.; Heizmann, M.: Ranking templates for linear loops. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014. pp. 172–186.

[13] Schrammel, P.; Kroening, D.: 2LS for Program Analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2016. pp. 905–907.