Methods for Proving Non-termination of Programs

Kaustubh Nimkar

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

May 7, 2015

I, Kaustubh Nimkar, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

The search for reliable and scalable automated methods for finding counterexamples to termination or alternatively proving non-termination is still widely open. The thesis studies the problem of proving non-termination of programs and presents new methods for the same. It also provides a thorough comparison of new methods along with the previous methods.

In the first method, we show how the problem of non-termination proving can be reduced to a question of underapproximation search guided by a safety prover. This reduction leads to new non-termination proving implementation strategies based on existing tools for safety proving. Furthermore, our approach leads to easy support for programs with unbounded non-determinism.

In the second method, we show how Max-SMT-based invariant generation can be exploited for proving non-termination of programs. The construction of the proof of non-termination is guided by the generation of quasi-invariants - properties such that if they hold at a location during execution once, then they will continue to hold at that location from then onwards. The check that quasi-invariants can indeed be reached is then performed separately. Our technique produces more generic witnesses of non-termination than existing methods. Moreover, it can handle programs with unbounded non-determinism and is more likely to converge than previous approaches.

When proving non-termination using known techniques, abstractions that overapproximate the program's transition relation are unsound. In the third method, we introduce live abstractions, a natural class of abstractions that can be combined with the concept of closed recurrence sets to soundly prove non-termination. To demonstrate the practical usefulness of this new approach we show how programs with non-linear, non-deterministic, and heap-based commands can be shown non-terminating using linear overapproximations.

All three methods introduced in this thesis have been implemented in different

tools. We also provide experimental results which show great performance improvements over existing methods.

Acknowledgements

First of all, I would like to thank my primary supervisor Byron Cook for introducing me to the beautiful world of program termination. His belief that I could succeed was very encouraging. His support during my PhD has been invaluable. He introduced me to many researchers along the way and that has been extremely beneficial.

I would also like to thank my secondary supervisor Peter O'Hearn for his support. His help in formalizing some of my ideas was crucial. I am extremely grateful to Carsten Fuhs for being a great collaborator, mentor and a very good friend. His keenness in many of my ideas and help during discussions have been very fruitful.

I am also thankful to my examiners, Andy King and James Brotherston for carefully reading the thesis. Their comments have greatly helped in adding a lot more clarity to the thesis.

I would like to thank all of my collaborators: Hongyi Chen, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell and Albert Rubio. It's been a great experience working with them.

I am also grateful to Robert Nieuwenhuis for giving me an opportunity to work in his group prior to my PhD. That experience was extremely encouraging and has helped me build a great level of confidence. I am also thankful to Supratik Chakraborty for introducing me to formal methods and verification. The time I spent at IIT Bombay working with him was very precious and motivated me to pursue a PhD.

I am also thankful to my parents, my younger brother, my parents-in-law and my cousin Teju for their support. Finally I am extremely thankful to my wife, Neha. Her constant support during my illness, medical treatment and beyond is just invaluable.

Contents

1 Introduction **11**

1.1 Non-termination – Key Notions 12

Nested Loops 13

Periodic vs Aperiodic Non-termination 14

1.2 Related Work 16

1.3 Contributions of this Thesis 19

Empirical Evaluation 20

Structure of the Thesis 20

2 Preliminaries **21**

2.1 Representing a Program using its Control Flow Graph (CFG) 23

CFG loops 25

2.2 Recurrence sets. 26

2.3 Summary 26

3 Proving Non-termination via Safety **27**

3.1 Informal Description of Our Method 27

3.2 Closed Recurrence Sets 30

Underapproximation 31

Preconditions 32

3.3 Our Procedure 32

Proof of Correctness 37

3.4 Advantages 39

3.5 Limitations 42

3.6 Summary and Outlook 42

4 Proving Non-termination Using Max-SMT **43**

4.1 Introduction . . . 43

4.2 Preliminaries . . . 46

SMT and Max-SMT . . . 46

Simplified CFGs . . . 47

4.3 Quasi-invariants and Non-termination . . . 49

4.4 Computing Proofs of Non-termination . . . 52

4.5 Advantages . . . 58

4.6 Limitations . . . 60

4.7 Summary and Outlook . . . 61

5 Proving Non-termination with Overapproximation **63**

5.1 Introduction . . . 63

5.2 Illustrating Example . . . 64

Combining over- and underapproximation . . . 66

5.3 Closed recurrence sets and overapproximation . . . 67

5.4 Live abstractions . . . 68

5.5 Classes of Live Abstractions for Automation . . . 71

Abstracting non-linear commands . . . 71

Dealing with non-linear guards . . . 72

Abstracting heap-based commands . . . 72

Combining over- and underapproximation . . . 73

5.6 Finding Closed Recurrence Sets . . . 74

5.7 Our Procedure . . . 79

5.8 Advantages . . . 80

Comparison with a method based on invariant . . . 80

5.9 Limitations . . . 80

5.10 Summary and Outlook . . . 81

6 Experiments **82**

6.1 Experimental Results on Programs with Linear Integer Arithmetic . . . 83

Discussion . . . 85

6.2 Experimental Results on Programs with Non-linear Integer Arithmetic and Heap based Commands 86
6.3 Summary 88

7 Conclusion **89**

Bibliography **90**

List of Figures

1.1 Nested Loops – Inner Loop Non-terminating 13

1.2 Nested Loops – Outer Loop Non-terminating 13

1.3 Aperiodic Non-termination - Single Loop 15

1.4 Aperiodic Non-termination - Nested Loops 15

2.1 A program in pseudo-code and its CFG 24

3.1 Example 27

3.2 Original instrumented program **(a)** and its successive underapproximations **(b)**, **(c)**, **(d)**. Reachability check for the loop **(e)**, and non-determinism-`assume` that must be checked for satisfiability **(f)**. 29

3.3 Procedure PROVER-SAFETY for underapproximation to synthesize a reachable non-terminating loop. To prove non-termination of P, PROVER-SAFETY should be run on all loops L. 33

3.4 Program with Repeating Counterexamples 34

3.5 Program showing why we need VALIDATE procedure 36

3.6 Instrumented program for the program of Figure 1.1 39

3.7 Instrumented program for the program of Figure 1.2 40

3.8 Instrumented program for the program of Figure 1.3 41

3.9 Instrumented program for the program of Figure 1.4 41

4.1 Example program (a) together with its corresponding CFG (b), non-trivial SCSGs (c) and non-termination analysis (d) 44

4.2 Program involving non-deterministic assignments **(a)**, and its simplified CFG **(b)** 49

4.3 Procedure PROVER-MAXSMT for proving non-termination of a program P by analyzing SCSG $\mathcal{C}$ 53

4.4 **(a)** Simplified CFG for program of Figure 1.1 and **(b)** the SCSG involved in non-termination . . . 59
4.5 **(a)** Simplified CFG for program of Figure 1.2 and **(b)** the SCSG involved in Non-termination . . . 59
4.6 **(a)** Simplified CFG for program of Figure 1.3 and **(b)** the SCSG involved in Non-termination . . . 60
4.7 **(a)** Simplified CFG for program of Figure 1.4 and **(b)** the SCSG involved in Non-termination . . . 61

5.1 Non-linear program **(a)**, and its linear abstraction **(b)**. . . . 64
5.2 Non-linear program **(a)**, its underapproximation **(b)**, and the resulting linear abstraction **(c)**. . . . 66
5.3 Linear overapproximation of the program in Figure 5.1(a) computed by our tool using APRON [JM09] . . . 72
5.4 Lasso **(a)** with non-linear guards and equivalent lasso **(b)** with auxiliary variable with linear guards . . . 73
5.5 Heap-based program **(a)** with precondition that p points to a nonempty cyclic list and linear overapproximation **(b)** computed by THOR [MTLT10] . . . 74
5.6 Non-linear lasso **(a)** and its live abstraction **(b)**. . . . 77
5.7 Our non-termination proving procedure PROVER-OVERAPPROX . . . 79

6.1 Evaluation success overview, showing the number of problems solved for each tool. Here **(a)** represents the results for known non-terminating examples, **(b)** is known terminating examples, **(c)** is (previously) unknown examples. . . . 84
6.2 Results ("Res") and runtimes of ANANT, APROVE, and JULIA on 29 benchmarks with non-linear arithmetic and 4 heap-based benchmarks from Berdine *et al.* [BCDO06]. Here ✓ denotes that the tool proved non-termination, × means that the tool returned without a definite answer, and *timeout* means that the run was terminated externally after 600 s. . . . 87

Chapter 1

Introduction

The problem of whether a given program will terminate on all possible inputs is a classical undecidable problem in computer science. This problem is not even semi-decidable [BMS05d]. Nevertheless, in recent years there has been a lot of research in devising sound, yet incomplete automated methods for proving program termination [FGKP85, DGG00, CS02, PR04, BMS05a, BMS05b, BMS05c, BMS05d, Cou05, CPR06, GST06, PR07, BHRC07, CGL+08, CPR09, OBEG10, KSTW10, TSWK11, BCF13, CSZ13, BCHR13, CKRW13]. This research has resulted in the development of many automatic termination proving tools that try to find a proof of termination for a given input program. The main focus of these tools is on finding proofs of termination and not on finding counterexamples to termination. To date only few tools use sophisticated methods to find counterexamples to termination.

A user of a verification tool is often interested in knowing that a given program either always obeys certain property or she is interested in finding a counterexample to the property. In case of termination provers, as the underlying methods are incomplete, failure to find a proof of termination does not imply existence of a counterexample. Thus termination provers need dedicated methods for finding counterexamples.

A counterexample to termination often represents a software bug and its existence is thus undesired. Automatically finding such counterexamples will be immensely useful to the programmer. The problem of finding counterexamples to termination is particularly interesting since such a counterexample represents an infinite execution of the original program and thus cannot be represented by a finite trace. Thus testing cannot be reliably used to identify them. To tackle this problem we need compact finite structures that capture infinite traces.

While the problem of proving termination has now been extensively studied, relatively less work has been done on finding counterexamples to termination. The search for reliable and scalable automated methods for finding such counterexamples is still widely open. The purpose of this thesis is to bridge this gap by providing some key contributions in the area of automated methods for finding counterexamples to termination or alternatively proving non-termination of programs. As the underlying problem is in general not even semi-decidable, the methods for proving non-termination (existing ones as well as those presented in this thesis) are only sound but incomplete. We now provide some key concepts related to non-termination, discuss related work and then explain the key contributions of this thesis.

1.1 Non-termination – Key Notions

A program is said to be non-terminating if there exists an infinite run of that program from some initial states. A method for proving non-termination analyses a program and upon finding a successful proof may or may not provide a witness to the user. A witness of non-termination can include various information like

- Exact initial states from which the program gets into an infinite run.
- Some closed structure (*e.g.* a loop) of a program's control flow graph to which an infinite run gets confined.
- An exact path from an initial program location to such closed structure that then results in an infinite run.
- A set of program states to which an infinite run gets confined.

Different program features can impose different challenges in proving non-termination. The main differentiating factors here depend on whether a program

- operates on integers or other data types
- treats integers as bounded (machine integers) or unbounded (mathematical integers)
- involves linear or non-linear arithmetic operations

```
    int i;
ℓ1 : if (i == 10) {
ℓ2 :   while (i > 0) {
ℓ3 :     i := i − 1;
ℓ4 :     while (i == 0)
ℓ5 :       skip;
ℓ6 :   }
ℓ7 : }
ℓ8 :
```

Figure 1.1: Nested Loops – Inner Loop Non-terminating

```
    int i, j;
ℓ1 : while (i > 0) {
ℓ2 :   i := i + 1;
ℓ3 :   j := 2;
ℓ4 :   while (j > 0)
ℓ5 :     j := j − 1;
ℓ6 : }
ℓ7 :
```

Figure 1.2: Nested Loops – Outer Loop Non-terminating

- is deterministic or non-deterministic
- includes commands that operate on the heap
- involves nested loops
- includes other program features like arrays, lists
- involves function calls, recursion

Different methods try to handle only some of these features. These methods thus can be differentiated based on the features they handle and also on the type of witness they provide to the user.

Nested Loops

Programs with nested loops pose some challenges in proving non-termination. We provide two simple examples which will be useful to facilitate discussion in section Section 1.2.

Example 1.1. Consider the program in Figure 1.1 with nested loops. Here the outer loop decreases the value of the variable i 10 times and then it is the inner loop that is non-terminating.

Example 1.2. Consider the program in Figure 1.2 with nested loops. The inner loop executes for exactly two iterations in each iteration of the outer loop. However the outer loop is non-terminating because it keeps incrementing the value of variable i and the loop condition is always met.

Periodic vs Aperiodic Non-termination

Another key notion is that of periodic vs aperiodic non-termination. A non-terminating run is said to be periodic if it visits the same fixed sequence of program locations, in succession and infinitely often. It is said to be aperiodic otherwise. An example of a periodic non-terminating run would be

$$\ell_0 \to \ell_1 \to \ell_2 \to \ell_3 \to \ell_1 \to \ell_2 \to \ell_3 \to \dots$$

where ℓ_0, ℓ_1, ℓ_2 and ℓ_3 are program locations. Here the fixed sequence of program locations $\ell_1 \to \ell_2 \to \ell_3$ is repeated in succession and infinitely often. An example of an aperiodic non-terminating run would be

$$\ell_0 \to \ell_1 \to \ell_2 \to \ell_1 \to \ell_2 \to \ell_2 \to \ell_1 \to \ell_2 \to \ell_2 \to \ell_2 \to \ell_1 \to \dots$$

Between every two successive ℓ_1 locations there is an increasing sequence of ℓ_2 locations and thus there is no fixed sequence of locations that is visited in succession and infinitely often.

Example 1.3. We revisit Example 1.1. The non-terminating run of this program is periodic. After the loop condition of the inner loop is met, the sequence of locations $\ell_4 \to \ell_5$ is visited in succession and infinitely often.

Example 1.4. We revisit Example 1.2. The non-terminating run of this program is periodic as well. The exact sequence of locations $\ell_1 \to \ell_2 \to \ell_3 \to \ell_4 \to \ell_5 \to \ell_4 \to \ell_5 \to \ell_4 \to \ell_6$ is visited in succession and infinitely often.

```
     int i, j, k;
ℓ1 : if (j ≥ 0) {
ℓ2 :   while(i ≥ j) {
ℓ3 :     k := i − j;
ℓ4 :     if (k > 0) {
ℓ5 :       i−−;
ℓ6 :     }
ℓ7 :     else {
ℓ8 :       i := 2 × i + 1;
ℓ9 :       j++;
ℓ10 :    }
ℓ11 :  }
ℓ12 : }
ℓ13 :
```

Figure 1.3: Aperiodic Non-termination - Single Loop

```
     int j, k;
ℓ1 : while (k ≥ 0) {
ℓ2 :   k := k + 1;
ℓ3 :   j := k;
ℓ4 :   while (j ≥ 1)
ℓ5 :     j := j − 1;
ℓ6 : }
ℓ7 :
```

Figure 1.4: Aperiodic Non-termination - Nested Loops

Aperiodic non-termination can arise from single as well as nested loops. We now provide the examples for the same.

Example 1.5. Consider the program in Figure 1.3 (taken from [BSOG12]). Here the loop does not terminate if we initially have $\mathsf{i} \geq \mathsf{j} \geq 0$. For example if $\mathsf{i} = 1, \mathsf{j} = 1$ at the beginning, then after one iteration of the loop we have $\mathsf{i} = 3, \mathsf{j} = 2$. In the next iteration we get $\mathsf{i} = 2, \mathsf{j} = 2$. In the next iterations we get $\mathsf{i} = 5, \mathsf{j} = 3$; $\mathsf{i} = 4, \mathsf{j} = 3$; $\mathsf{i} = 3, \mathsf{j} = 3$. Whenever i equals j, after the `else` branch we again get $\mathsf{i} \geq \mathsf{j}$ and thus the loop does not terminate. Moreover between every two successive visits to the `else` branches, the `if` branch is visited increasingly many times. Thus there is no fixed sequence of program locations that is visited in succession and infinitely often. Thus the non-terminating runs of this program are aperiodic.

Example 1.6. Consider the program in Figure 1.4 with nested loops. In every iteration of the outer loop the inner loop iterates but always terminates. However if $\mathsf{k} \geq 0$ at the beginning then the outer loop simply increments k and thus its loop condition is always satisfied. Thus the program is non-terminating. Suppose for any chosen iteration of the outer loop the inner loop makes n iterations. For the next iteration of the outer loop the inner loop makes exactly $n + 1$ iterations and this repeats forever. Thus there is no fixed sequence of locations that is visited in succession and infinitely often. Thus the non-terminating runs of this program are aperiodic.

1.2 Related Work

While in some trivial cases termination provers can easily disprove termination (*e.g.* when variables are not modified in an infinite loop), in practice this is not the focus for these tools.

Gupta *et al.* [GHM+08] use a characterisation of non-termination by (open) recurrence sets (See Section 2.2). A recurrence set exists *iff* a program is non-terminating. To find recurrence sets they provide a method based on constraint solving. Their method is restricted to programs using linear integer arithmetic and does not support non-determinism. Additionally the method finds witnesses of non-termination in form of simple lasso-shaped paths where the same sequence of commands in every iteration of the loop leads to non-termination. Thus their method cannot detect aperiodic non-termination arising out of single as well as nested loops. Their method can possibly handle nested loops if the non-termination is periodic. For example, their method can prove non-termination of Example 1.1 and Example 1.2, but it cannot prove non-termination of Example 1.5 and Example 1.6. The method is implemented in the tool TNT. Given a program, TNT exhaustively enumerates lassos present in the program and checks each lasso for a proof of non-termination. As the number of lassos present in any program with a loop are infinite, TNT may diverge in this process. Thus it may not be able to detect a non-terminating lasso even if one exists in the program. TNT can technically handle programs with machine integers but for such programs it can only detect singleton recurrence sets (*i.e.* the same program state is visited infinitely often). In order to find more generic recurrence sets, it needs a restriction to programs with unbounded integers.

APROVE [GBE+14] uses SMT solving to prove non-termination of Java programs [BSOG12]. First non-termination of a loop regardless of its context is proved, then reachability of the loop with suitable values. The drawback of their technique is that they require either singleton recurrence sets or that the loop conditions themselves must be recurrence sets. Their technique in general supports unbounded integers, other data types, heap based commands and nested loops but for these generic features they can only detect non-termination with singleton recurrence sets. Their technique can find generic recurrence sets only in case of unbounded integers with single loops (without any subloops). The reason for this limitation is the following: their technique requires enumeration of all paths in the loop. This cannot be achieved when there is a subloop as there can be infinitely many paths inside the outer loop. Thus their technique can possibly detect non-termination of Example 1.1 as the non-terminating loop (the inner loop) does not have any subloops. However it cannot detect non-termination of Example 1.2. Their technique can detect aperiodic non-termination when it comes from a single loop and cannot detect aperiodic non-termination associated with nested loops. Thus their technique can possibly detect non-termination of Example 1.5, but not of Example 1.6.

Velroyen *et al.* [VR08] present a method for non-termination proving of Java programs using a combination of theorem proving and generation of plausible invariants. The method is mainly based on heuristics where a number of plausible invariants are generated and then for each of them a check is made if the proof tree can be closed. If the proof tree can be closed then it proves that the invariant is real and the program is non-terminating. Their method is implemented in the tool INVEL. INVEL is only applicable to deterministic programs with unbounded integers and can only handle single loops. The authors claim that their method can be extended in theory to other program features and nested loops. It is not clear if their technique can detect aperiodic non-termination.

Non-termination analysis tools for constraint-logic programs (*e.g.* [PM09]) can in cases be used to prove non-termination of imperative programs (*e.g.* JULIA [PS09] can show non-termination for Java bytecode programs if the abstraction to constraint-logic programs is exact, but does not provide a witness like a recurrence set to the user). The main difficulty here is in the application of the tools to imperative programs,

as overapproximating abstractions are typically used for converting languages such as Java and C to *e.g.* constraint-logic programs. These abstractions are in general unsound for directly proving non-termination. Our contributions in Chapter 5 of this thesis may in fact have application in this domain.

Gulwani *et al.* [GSV08] can prove non-termination in some cases by proving the exit points of the program unreachable, but use a restriction to linear arithmetic. Their technique is fairly imprecise in the presence of non-determinism in the input.

Atig *et al.* [ABEL12] reduce non-termination of multithreaded programs to non-termination reasoning for sequential programs. Our work complements Atig *et al.*, as we improve the underlying sequential tools that future multithreaded tools can use.

Tools for proving CTL properties of infinite state programs exist. These tools can mainly prove CTL properties with universal path quantifiers (*i.e.* properties that are valid for all computation paths), *e.g.*, [CKV11], [CCG+05]. Some recent approaches (*e.g.* [CK13], [CKP14], [GWC06]) can successfully reason about CTL properties with existential path quantifiers (*e.g.* proving whether there exists a particular computation path). Non-termination is an existential property and can be expressed in CTL as $\mathsf{EG}\ pc \neq \mathsf{END}$. The tool YASM [GWC06] can prove non-termination in some cases. The tool is no longer supported and during our experiments we were unable to obtain a running version of YASM to compare against. The techniques in [CK13], [CKP14] are based on the idea that existential reasoning can be reduced to universal reasoning if the program's state space is appropriately restricted. These techniques require that the restricted state space is a recurrence set and thus need an off-the-shelf non-termination prover to prove existence of a recurrence set. The work presented in this thesis may in fact have applications in this domain.

Finally, several automatic tools exist for proving non-termination of term rewrite systems (*e.g.* [EEG12], [GTS05], [Pay08]). However, in non-termination analysis for term rewriting the *entire* state space is considered as legitimate initial states for a (possibly infinite) evaluation sequence, whereas our setting also factors in reachability from the initial states.

1.3 Contributions of this Thesis

This thesis presents new methods for proving non-termination. The new methods try to handle program features which previous methods either could not handle or had very limited support for. The contributions of this thesis can be summarized as follows.

(I) Chapter 3 provides a new method for proving non-termination which reduces the problem of non-termination proving to underapproximation search guided by a safety prover. This reduction leads to new non-termination proving implementation strategies based on existing tools for safety proving. This approach also leads to easy support for programs with unbounded non-determinism which is explored in detail in the chapter. Previous tools either did not support or had very little support for non-determinism. In this chapter, we introduce closed recurrence sets which extend the concept of recurrence sets [GHM+08]. We show that our desired underapproximation contains a closed recurrence set as a witness of non-termination.

The new method can also handle nested loops and detect aperiodic non-termination arising from single as well as nested loops.

This contribution has also been published in the conference paper [CCF+14]. The method is implemented in the tool T2[1]. This implementation only considers programs with linear arithmetic commands on unbounded integers. However the technique could be extended to support other programming language features.

(II) Chapter 4 provides a new method for proving non-termination based on Max-SMT-based invariant generation. This method considers strongly connected subgraphs of a program's control flow graph for analysis and thus produces more generic witnesses of non-termination than existing methods. Moreover, the method can also handle programs with unbounded non-determinism and is more likely to converge (*i.e.* it terminates with a successful proof of non-termination) than other approaches. The method can also handle nested loops and detect aperiodic non-termination arising from single as well as nested loops.

This contribution has also been published in the conference paper [LNO+14].

[1] The implementation was done by a coauthor of [CCF+14].

The method is implemented in the tool CPPINV[2]. This implementation only considers programs with linear arithmetic commands on unbounded integers.

(III) When proving non-termination using known techniques (*e.g.* recurrence sets), abstractions that overapproximate the program's transition relation are unsound. In Chapter 5 we introduce *live abstractions*, a natural class of abstractions that can be combined with the concept of *closed recurrence sets* (introduced in Chapter 3) to soundly prove non-termination. To our knowledge, this is the first method which uses overapproximation for proving non-termination. To demonstrate the practical usefulness of this new approach we show how programs with non-linear, non-deterministic and heap-based commands can be shown non-terminating using linear overapproximations. The method is implemented in the tool ANANT which is developed completely by the author of this thesis.

This contribution has also been published in the conference paper [CFNO14].

Empirical Evaluation

As described previously the new non-termination proving methods introduced in Chapter 3, Chapter 4 and Chapter 5 have been implemented in the tools T2, CPPINV and ANANT respectively. In Chapter 6 we compare these tools against each other and also provide a detailed comparison with the existing tools for proving non-termination. Our experiments show that the new methods provide great performance improvements over the existing methods. Particularly T2 and CPPINV are overwhelmingly successful when the input programs contain non-determinism. ANANT is overwhelmingly successful when the input programs contain non-linear arithmetic or heap-based commands.

Structure of the Thesis

In Chapter 2 we provide the main definitions and basic notations that are used throughout the thesis. As discussed, Chapter 3, Chapter 4 and Chapter 5 describe the three new non-termination proving methods. Chapter 6 provides the experimental results and in Chapter 7 we conclude and discuss some future research directions.

[2]The implementation was done by a coauthor of [LNO+14].

Chapter 2

Preliminaries

We now provide the main definitions and basic notations that are used throughout the thesis.

Definition 2.1 (Transition Systems)**.** A *transition system* (S, R, I, F) is defined by a set of states S, a *transition relation* $R \subseteq S \times S$, a set of *initial states* $I \subseteq S$ and a set of *final states* $F \subseteq S$. For a state s with $R(s, s')$, we say that s' is a *post-state* of s and that s is a *pre-state* of s'. We also call s' a *successor of* s *under* R. An execution of a transition system is a sequence of states s_0, s_1, s_2 ... such that $s_0 \in I$ and for each pair of consecutive states, we have $R(s_j, s_{j+1})$. Any execution of a transition system can only halt in a final state, so every state $s \notin F$ must have a successor under R, and any final state $f \in F$ has no successors under R. Formally, $F = \{s \mid s \in S \wedge \neg \exists s'.\ R(s, s')\}$.

For a set of states $\mathcal{G}$ and a state s, we write $\mathcal{G}(s)$ to represent $s \in \mathcal{G}$.

Definition 2.2 (Programs)**.** A program is a tuple $(\overline{\mathsf{v}}, \mathcal{L}, \mathcal{T}, \Theta)$ where $\overline{\mathsf{v}}$ is a tuple of *program variables*, $\mathcal{L}$ is a set of *locations*, and $\mathcal{T}$ is a set of *transitions* and Θ is a formula in first order logic over $\overline{\mathsf{v}}$ representing the program's precondition.[1] Each transition $\tau \in \mathcal{T}$ is a triple (ℓ, ℓ', T), where ℓ, $\ell' \in \mathcal{L}$ are the *pre* and *post* locations respectively, and T is a program command which must be one of the following.

- A `skip` command that does not modify values of the program variables.
- A deterministic assignment statement of the form $\mathsf{i} := \mathsf{exp}$ where $\mathsf{i} \in \overline{\mathsf{v}}$ and exp is an expression over program variables $\overline{\mathsf{v}}$.[2]

[1] The intuition behind this definition is to be able to represent a program using its control flow graph.

[2] Depending on the non-termination proving method under consideration, we specify which expressions are allowed.

- A non-deterministic assignment statement of the following form: i := `nondet()`; `assume`(Q); where $\mathsf{i} \in \overline{\mathsf{v}}$, `nondet()` is a non-deterministic choice and Q is a boolean expression over $\overline{\mathsf{v}}$ representing the restriction that the `nondet()` choice must obey.

- A conditional statement encoded using an `assume` command (from Nelson [Nel89]): `assume`(Q), where Q is a boolean expression over $\overline{\mathsf{v}}$.[3] We require that if there is a transition (ℓ, ℓ', `assume`(Q)), there is a transition (ℓ, ℓ'', `assume`($\neg\, \mathsf{Q}$)) with ℓ'' being the only other post location of ℓ.

We represent by $\ell_{\mathcal{I}} \in \mathcal{L}$ the initial location of a program. We represent by $\ell_{\mathcal{F}} \in \mathcal{L}$ the final location of a program. The final location does not have any outgoing transitions.[4] We represent by $\ell_{\mathcal{E}} \in \mathcal{L}$ the error location of a program. For the sake of presentation, we assume that the non-determinism of programs can come only from non-deterministic assignments of the form $\mathsf{i} :=$ `nondet()`, where $\mathsf{i} \in \overline{\mathsf{v}}$ is a program variable. Note that, however, this assumption still allows one to encode other kinds of non-determinism. For instance, any non-deterministic branching of the form **if**$(*)\{\}$ **else**$\{\}$ can be cast into this framework by introducing a new program variable $\mathsf{k} \in \overline{\mathsf{v}}$ and rewriting into the form k := `nondet()`; **if**$(\mathsf{k} \geq 0)\{\}$ **else**$\{\}$.

Definition 2.3 (Memory States and Program States)**.** A *memory state* σ is defined over a tuple of variables $\overline{\mathsf{v}}$ and consists of an assignment of a value to each of the variables in $\overline{\mathsf{v}}$. We represent by $\mathcal{M}$, the set of all memory states. For instance, for a program on n integer variables, we have $\mathcal{M} = \mathbb{Z}^n$. For a memory state σ and a formula Ψ, we write $\sigma \models \Psi$ when σ satisfies Ψ. For a transition (ℓ, ℓ', T) we represent by $\mathcal{R}_T \subseteq \mathcal{M} \times \mathcal{M}$, the relation on memory states induced by the command T in the usual way. We sometimes abuse the notation $\mathcal{R}_T$ slightly and use it instead to represent the formula for the transition relation induced by the command T. This formula $\mathcal{R}_T$ is over the program variables $\overline{\mathsf{v}}$ and their primed versions $\overline{\mathsf{v}}'$, which represent the values of the variables after the transition. For memory state σ defined over $\overline{\mathsf{v}}$, memory state σ' defined over $\overline{\mathsf{v}}'$ and a formula $\mathcal{R}_T$, we write $(\sigma, \sigma') \models \mathcal{R}_T$ when (σ, σ') satisfy $\mathcal{R}_T$.

[3] The `assume` command does not allow executions to pass unless the condition Q holds. For termination, we can encode **assume**(Q) $\equiv$ **if** $\neg\, \mathsf{Q}$ **then exit**(); **fi**

[4] Without loss of generality, we assume that there is a unique final location, since all locations without any outgoing transitions can be represented by a unique final location.

A *program state* is a pair (ℓ, σ) consisting of a location $\ell \in \mathcal{L}$ and a memory state σ. For any pair of program states (ℓ, σ) and (ℓ', σ'), if there is a transition $\tau = (\ell, \ell', T) \in \mathcal{T}$ such that $(\sigma, \sigma') \models \mathcal{R}_T$, we write $(\ell, \sigma) \xrightarrow{\tau} (\ell', \sigma')$.

Definition 2.4 (Computation)**.** A *computation* is a sequence of program states (ℓ_0, σ_0), $(\ell_1, \sigma_1) \ldots$ such that $\ell_0 = \ell_{\mathcal{I}}$ and $\sigma_0 \models \Theta$, and for each pair of consecutive program states there exists $\tau_j \in \mathcal{T}$ satisfying $(\ell_j, \sigma_j) \xrightarrow{\tau_j} (\ell_{j+1}, \sigma_{j+1})$. A program state (ℓ, σ) is *reachable* if there exists a computation which contains (ℓ, σ). A location $\ell \in \mathcal{L}$ is *reachable* if a program state (ℓ, σ) is reachable for some memory state σ. A program is *terminating* if all its computations are finite, and *non-terminating* otherwise. A program is said to be safe if $\ell_{\mathcal{E}}$ is not reachable and unsafe otherwise.

Definition 2.5 (Program as a Transition System)**.** Any program (or program fragment) $(\overline{\mathsf{v}}, \mathcal{L}, \mathcal{T}, \Theta)$ can be represented as a transition system (S, R, I, F) where $S = \mathcal{L} \times \mathcal{M}$, $I = \{(\ell_{\mathcal{I}}, \sigma) \mid \sigma \models \Theta\}$, $R = \{((\ell_j, \sigma_j), (\ell_{j+1}, \sigma_{j+1})) \mid \exists \tau_j \in \mathcal{T}.(\ell_j, \sigma_j) \xrightarrow{\tau_j} (\ell_{j+1}, \sigma_{j+1})\}$, $F = \{(\ell, \sigma) \mid (\ell, \sigma) \in S \wedge \neg\exists(\ell', \sigma').\ R((\ell, \sigma), (\ell', \sigma'))\}$.

Definition 2.6 (Underapproximation)**.** We call a transition system (S, R', I', F') an underapproximation of a transition system (S, R, I, F) *iff* $R' \subseteq R$, $I' \subseteq I$, $F \subseteq F'$.

2.1 Representing a Program using its Control Flow Graph (CFG)

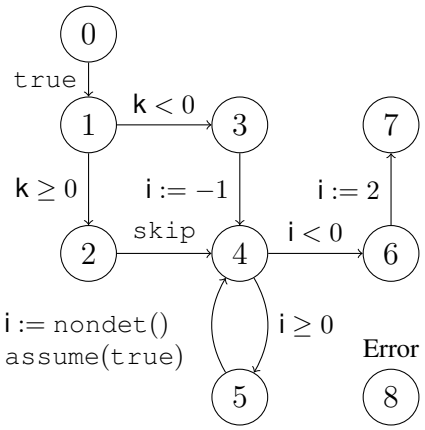
We often represent a program $(\overline{\mathsf{v}}, \mathcal{L}, \mathcal{T}, \Theta)$ via its CFG. Every node of the CFG represents a location $\ell \in \mathcal{L}$. For every transition $(\ell, \ell', T) \in \mathcal{T}$, there exists a corresponding directed edge from node ℓ to node ℓ' in the CFG which is labelled with T. Thus depending on the context, we use the notation (ℓ, ℓ', T) to represent either the transition or the edge in the CFG. Similar to Definition 2.2, by $\ell_{\mathcal{I}}$, $\ell_{\mathcal{F}}$ and $\ell_{\mathcal{E}}$, we designate special nodes of the CFG where $\ell_{\mathcal{I}}$ is the node representing the initial location, $\ell_{\mathcal{F}}$ is the node representing the final location and $\ell_{\mathcal{E}}$ is the node representing the error location. If we have $(\ell, \sigma) \xrightarrow{\tau} (\ell', \sigma')$ for any transition $\tau \in \mathcal{T}$, then we say that (ℓ', σ') is a successor of (ℓ, σ) along the edge representing τ in the CFG. A path π in the CFG is a sequence of edges $(\ell_0, \ell_1, T_0)\ (\ell_1, \ell_2, T_1)\ \ldots\ (\ell_{n-1}, \ell_n, T_{n-1})$. The composite transition relation $\mathcal{R}_\pi$ of the path π is the composition $\mathcal{R}_{T_0} \circ \mathcal{R}_{T_1} \circ \ldots \circ \mathcal{R}_{T_{n-1}}$ of the

```
if (k ≥ 0)
  skip;
else
  i := -1;

while (i ≥ 0) {
  i := nondet();
}

i := 2;
```

(a) Simple Program



(b) Program's Control Flow Graph

Figure 2.1: A program in pseudo-code and its CFG

individual relations. We also describe path π by the sequence of nodes it visits, e.g. $\ell_0 \to \ell_1 \to \ldots \to \ell_{n-1} \to \ell_n$. For readability, in our example CFGs we often write Q instead of $\texttt{assume}(\mathsf{Q})$. Without loss of generality, we assume $\ell_{\mathcal{I}}$ has 0 incoming and 1 outgoing edge, labeled with $\texttt{assume}(\Theta)$, where Θ represents the program's precondition. Additionally $\ell_{\mathcal{F}}$ does not have any outgoing edge.

We define the indegree of a node ℓ in a CFG to be the number of incoming edges to ℓ. Similarly, the outdegree of a node ℓ in a CFG is the number of outgoing edges from ℓ. A node $\ell \in \mathbb{L} \setminus \{\ell_{\mathcal{I}}, \ell_{\mathcal{F}}, \ell_{\mathcal{E}}\}$ must be of one of the following types.

1. A deterministic assignment node: ℓ has outdegree exactly 1 and the outgoing edge is labeled with a deterministic assignment statement or `skip`. Any program state (ℓ, σ) has a unique successor (ℓ', σ') along the edge.

2. A deterministic conditional node: ℓ has outdegree 2 with one edge labeled $\texttt{assume}(\varphi)$, the other edge labeled $\texttt{assume}(\neg\varphi)$. A program state (ℓ, σ) has a unique successor (ℓ', σ') along one edge and no successor along the other edge.

3. A non-deterministic assignment node: ℓ has outdegree exactly 1 and the outgoing edge is labeled with a non-deterministic assignment statement. A program state (ℓ, σ) may have zero or more successors along the outgoing edge depending on the condition present in the $\texttt{assume}(\mathsf{Q})$ statement.

Example 2.1. Consider a simple program in pseudo-code and its CFG from Figure 2.1. Here we have the initial location $\ell_{\mathcal{I}} = 0$, the final location $\ell_{\mathcal{F}} = 7$, and the error location $\ell_{\mathcal{E}} = 8$. The nodes 2, 3 and 6 are deterministic assignment nodes, nodes 1 and 4 are deterministic conditional nodes, and node 5 is a non-deterministic assignment node.

CFG loops

Given a program with its CFG, a loop L in the CFG is a set of nodes such that

- There exists a path from any node of L to any other node of L. In other words, the subgraph of the CFG containing all nodes of L is strongly connected.
- (W.l.o.g.) we assume that there is only one node h of L s.t. there exists a node $n \notin L$ with an edge from n to h. The node h is called the *header node* of L.

The subgraph of CFG containing all nodes of L is called the *loop body* of L. Header h of L is a deterministic conditional node with one edge that is part of the loop body, the *guard edge* of L. The other edge of h goes to a node $e \notin L$. We call this edge the *exit edge* of L and e the *exit location* of L.

Example 2.2. In Figure 2.1, the only loop is $L = \{4, 5\}$, and its header node h is 4. The exit location of L is 6.

Definition 2.7 (Loop Path)**.** Given a loop L in program P, we define a *loop path* π_L as any finite path through L's body of the form $(\ell_0 = h) \to \ell_1 \to ... \to \ell_{n-1} \to (\ell_n = h)$, where h is the header node of L and $\forall p.(0 < p < n) \to \ell_p \neq h$.

Definition 2.8 (Loop as a Transition System)**.** We often analyze a loop L in a program P as a transition system (S_L, R_L, I_L, F_L). We analyze a loop as a single body of instructions with the loop header as a unique location. As there is a single program location, Definition 2.5 can be simplified by dropping program locations. Formally, $S_L = \mathcal{M}$, the memory states in P. We define the *composite transition relation* R_L of L as $R_L(\sigma, \sigma')$ *iff* there exists a loop path π s.t. $R_\pi(\sigma, \sigma')$. Here R_L need not be a non-empty finite set. It can be an infinite or an empty set. The initial states I_L for R_L are the set of reachable states at the loop header before the loop is entered for the first time and $F_L = \{\sigma \mid \sigma \in S_L \wedge \neg\exists\sigma'.\ R_L(\sigma, \sigma')\}$.

Chapter 4 and Chapter 5 provide methods for proving non-termination based on constraint solving, which make use of the following theorem from linear algebra.

Theorem 2.1 (Farkas' Lemma)**.** *Let S be a system of linear inequalities $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ ($\boldsymbol{A} \in \mathbb{R}^{m\times n}, \boldsymbol{b} \in \mathbb{R}^m$) over variables $\boldsymbol{x}^T = [x_1\ x_2\ \ldots\ x_n]$. When S is satisfiable, it entails a linear inequality $\boldsymbol{c}^T\boldsymbol{x} \leq d$ ($\boldsymbol{c} \in \mathbb{R}^n, d \in \mathbb{R}$) iff there is $\boldsymbol{\lambda} \in \mathbb{R}^m$ such that $\boldsymbol{\lambda} \geq \boldsymbol{0}$, $\boldsymbol{\lambda}^T\boldsymbol{A} = \boldsymbol{c}^T$ and $\boldsymbol{\lambda}^T\boldsymbol{b} \leq d$. Further, S is unsatisfiable iff $1 \leq 0$ can be so derived.* □

2.2 Recurrence sets.

A transition system (S, R, I, F) is non-terminating *iff* there exists an infinite transition sequence $s_0 \xrightarrow{R} s_1 \xrightarrow{R} s_2 \xrightarrow{R} \ldots$ with $s_0 \in I$. Gupta *et al.* [GHM+08] characterize non-termination of a relation R by the existence of a *recurrence set*, *viz.* a nonempty set of states $\mathcal{G}$ such that for each $s \in \mathcal{G}$ there exists a transition to some $s' \in \mathcal{G}$. In particular, an infinite transition sequence $s_0 \xrightarrow{R} s_1 \xrightarrow{R} s_2 \xrightarrow{R} \ldots$ itself gives rise to the recurrence set $\{s_0, s_1, s_2, \ldots\}$. In this work we extend the notion of a recurrence set to transition systems. A transition system (S, R, I, F) has a *recurrence set* of states $\mathcal{G}$ *iff* (2.1) and (2.2) hold.

$$\exists s.\mathcal{G}(s) \wedge I(s) \tag{2.1}$$

$$\forall s \exists s'.\mathcal{G}(s) \rightarrow R(s, s') \wedge \mathcal{G}(s') \tag{2.2}$$

Theorem 2.2. *[GHM+08] A transition system (S, R, I, F) is non-terminating* iff *it has a recurrence set of states.*

The notion of a recurrence set is a central idea in non-termination proving. Key contributions in this thesis are built upon this notion. To differentiate clearly with the notion of *closed recurrence set* (defined later in Section 3.2) we sometimes refer to a recurrence set as an *open recurrence set*.

2.3 Summary

We have described the main definitions and basic notations that are used in the rest of the thesis. We have also described the notion of a recurrence set, upon which we build the main contributions of this thesis.

Chapter 3

Proving Non-termination via Safety

In this chapter we develop a new method of proving non-termination based on a reduction to safety proving that leverages the power of existing tools. An iterative method is developed which uses counterexamples to a fixed safety property to refine an underapproximation of a program. With our approach, existing safety provers can now be employed to prove non-termination of programs that previous techniques could not handle. Not only does the new approach perform better, it also leads to non-termination proving tools supporting programs with non-determinism, for which previous tools had only little support.

3.1 Informal Description of Our Method

Before discussing our method in a formal setting, we first explain it informally using an example. Consider a simple program from Figure 3.1.

In this program the command i := nondet() represents non-deterministic value

```
int i, k;
if (k ≥ 0)
  skip;
else
  i := -1;

while (i ≥ 0) {
  i := nondet();
}

i := 2;
```

Figure 3.1: Example

introduction into the variable i. The loop in this program is non-terminating when the program is invoked with appropriate inputs and when appropriate choices for `nondet` assignment are made. We are interested in automatically detecting this non-termination. The basis of our method is the search for an underapproximation of the original program that *never* terminates. As "never terminates" can be encoded as safety property (defined later as *closed recurrence* in Section 3.2), we can then iterate a safety prover together with a method of underapproximating based on counterexamples. We have to be careful, however, to find the right underapproximation in order to avoid unsoundness.

In order to find the desired underapproximation for our example, we introduce an `assume` statement at the beginning with the initial precondition `true`. We also place `assume(true)` statements after each use of `nondet`. We then put an `assert(false)` statement at points where the loop under consideration exits (thus encoding the "never terminates" property). See Figure 3.2(a).

We can now use a safety checker to search for paths that violate this assertion. Any error path clearly cannot contribute towards the non-termination of the loop. After detecting such a path we calculate restrictions on the introduced `assume` statements such that the path is no longer feasible when the restriction is applied.

Initially as a first counterexample to safety, we might get the path $\mathsf{k} < 0,\ \mathsf{i} := -1,\ \mathsf{i} < 0$, from a safety prover. We now want to determine from which states we can reach `assert(false)` and eliminate those states. Using a precondition computation similar to Calcagno *et al.* [CDOY11] we find the condition $\mathsf{k} < 0$. The trick is to use the standard weakest precondition rule for assignments, but to use $pre(\texttt{assume}(Q), P) \triangleq P \wedge Q$ instead of the standard $wp(\texttt{assume}(Q), P) \triangleq Q \Rightarrow P$. This way, we only consider executions that actually reach the error location. To rule out the states $\mathsf{k} < 0$ we can add the negation (*e.g.* $\mathsf{k} \geq 0$) to the precondition `assume` statement. See Figure 3.2(b).

In our procedure we try again to prove the assertion statement unreachable, using the program in Figure 3.2(b). In this instance we might get the path $\mathsf{k} \geq 0,\ \texttt{skip},\ \mathsf{i} < 0$, which again violates the assertion. For this path we would discover the precondition $\mathsf{k} \geq 0 \wedge \mathsf{i} < 0$, and to rule out these states we refine the precondition assume statement with "`assume`($\mathsf{k} \geq 0 \wedge \mathsf{i} \geq 0$);". See Figure 3.2(c).

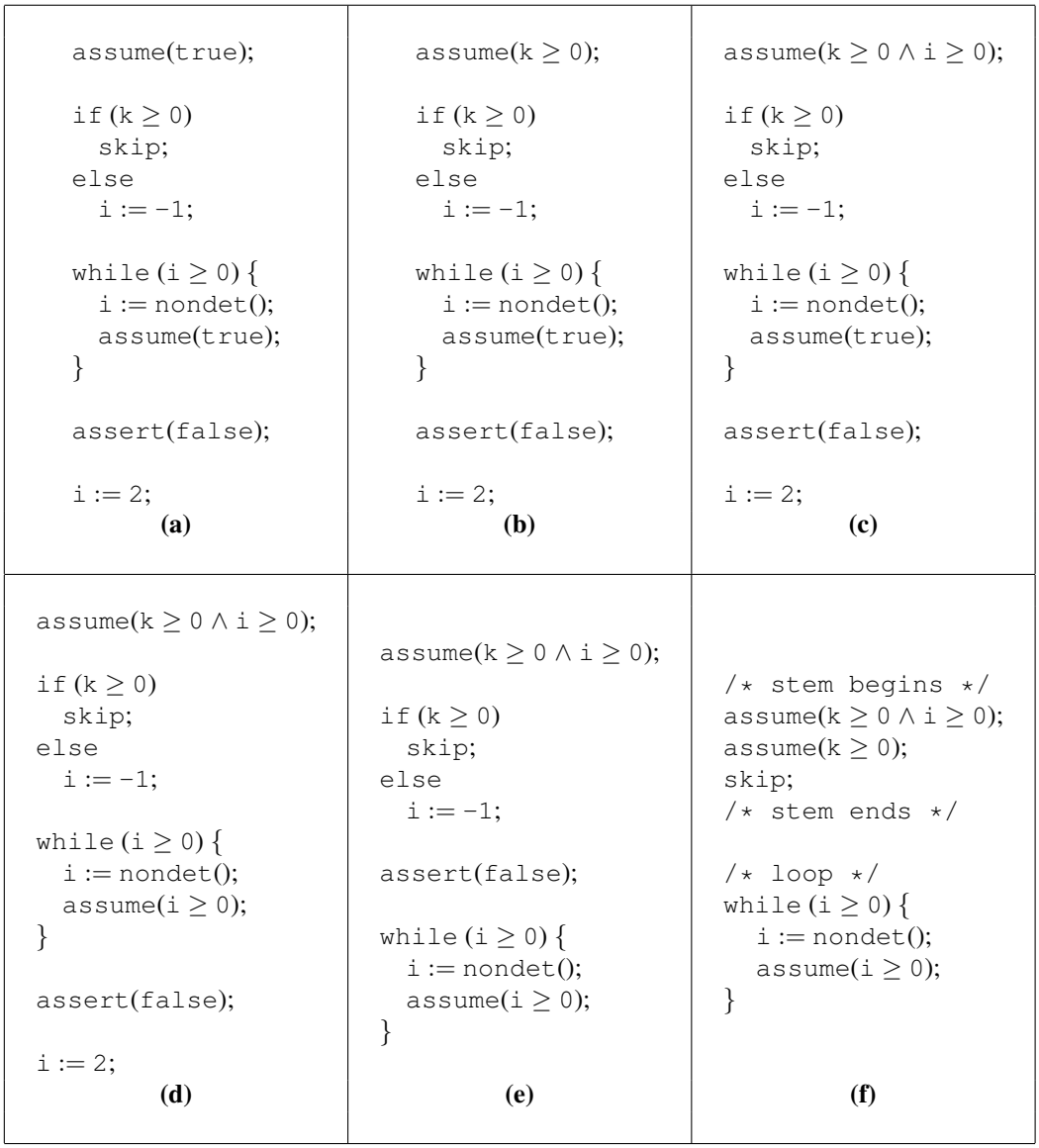


Figure 3.2: Original instrumented program **(a)** and its successive underapproximations **(b)**, **(c)**, **(d)**. Reachability check for the loop **(e)**, and non-determinism-`assume` that must be checked for satisfiability **(f)**.

On this program our safety prover will again fail, perhaps resulting in the path $\mathsf{k} \geq 0,\ \texttt{skip},\ \mathsf{i} \geq 0,\ \mathsf{i} := \texttt{nondet}(),\ \mathsf{i} < 0$. Then our procedure would stop computing the precondition at the command $\mathsf{i} := \texttt{nondet}()$ (for reasons discussed later). Here we would learn that at the non-deterministic command the result must be $\mathsf{i} < 0$ to violate the assertion, thus we would refine the `assume` statement just after the `nondet` with the negation of $\mathsf{i} < 0$: "$\texttt{assume}(\mathsf{i} \geq 0);$" See Figure 3.2(d).

The program in Figure 3.2(d) cannot violate the assertion, and thus we have hopefully computed the desired underapproximation to the transition relation needed in order to prove non-termination. However, for soundness, it is essential to ensure that the

loop in Figure 3.2(d) is still reachable, even after the successive restrictions to the state-space. We encode this condition as a safety problem. See Figure 3.2(e). This time we add `assert(false)` before the loop and aim to prove that the assertion is violated. The existence of a path violating the assertion ensures that the loop in Figure 3.2(d) is reachable. Here the assertion and thus the loop are still reachable. The path violating the assertion is our desired path to the loop which we refer to as *stem*. Figure 3.2(f) shows the stem and the loop.

Finally we need to ensure that the `assume` statement in the loop of Figure 3.2(f) can always be satisfied with some i by any reachable state from the restricted pre-state. This is necessary: since our underapproximations may accidentally have eliminated not only the paths to the loop's exit location, but also all of the non-terminating paths inside the loop. Once this check succeeds we have proved non-termination.

3.2 Closed Recurrence Sets

In this section we define a new concept which is at the heart of our method, called *closed recurrence*. Closed recurrence extends the concept of open recurrence from Section 2.2 in a way that facilitates automation, e.g. via a safety prover.

$$\exists s.\mathcal{G}(s) \wedge I(s) \tag{3.1}$$

$$\forall s \exists s'.\mathcal{G}(s) \rightarrow R(s, s') \tag{3.2}$$

$$\forall s \forall s'.\mathcal{G}(s) \wedge R(s, s') \rightarrow \mathcal{G}(s') \tag{3.3}$$

A set $\mathcal{G}$ is a *closed recurrence set* for a transition system (S, R, I, F) *iff* the conditions (3.1)–(3.3) hold. Notice how, in contrast to open recurrence sets, we now require a purely universal property: for each $s \in \mathcal{G}$ and *for each* of its successors s', also s' must be in the recurrence set (Condition (3.3)). So instead of requiring that we *can* stay in the recurrence set, we now demand that we *must* stay in the recurrence set. This now helps us to incorporate non-deterministic transition systems too.

There is an additional problem: what if a state s in our recurrence set $\mathcal{G}$ has no successor s' at all? This would bring our alleged infinite transition sequence to a sudden halt, yet our universal formula would trivially hold. To deal with this issue, we

must impose that each $s \in \mathcal{G}$ has *some* successor s' (Condition (3.2)). But this existential statement need not mention that s' must be in $\mathcal{G}$ again—our previous *universal* statement already takes care of this.

Theorem 3.1 (A Closed Recurrence Set is an Open Recurrence Set)**.** *Let $\mathcal{G}$ be a closed recurrence set for a transition system (S, R, I, F). Then $\mathcal{G}$ is also an open recurrence set for (S, R, I, F).*

Proof. We only need to show Condition (2.2), which follows directly from Condition (3.2) and Condition (3.3). □

If our transition system is *deterministic*, every open recurrence set is also a closed recurrence set. In particular, closed recurrence sets characterize non-termination in the setting of Gupta *et al.* [GHM+08], which assumes deterministic programs.

Theorem 3.2 (An Open Recurrence Set is a Closed Recurrence Set for a Deterministic Transition System)**.** *Let $\mathcal{G}$ be an open recurrence set for (S, R, I, F) such that for every state s there exists at most one state s' with $R(s, s')$. Then $\mathcal{G}$ is also a closed recurrence set for (S, R, I, F).*

Proof. We only need to show Condition (3.2) and Condition (3.3) for $\mathcal{G}$. Condition (3.2) is implied by Condition (2.2). For Condition (3.3): Let s, s' such that $\mathcal{G}(s)$ and $R(s, s')$. Since R is deterministic, we have $\neg\exists s''.\ R(s, s'') \wedge s' \neq s''$. Thus Condition (2.2) gives us $\mathcal{G}(s')$. Thus we have Condition (3.3) for $\mathcal{G}$. □

Underapproximation

We now state and prove the main theorem that gives the rationale behind our procedure. Referring to Definition 2.6 we prove that *every* non-terminating program contains a closed recurrence set as an underapproximation (*i.e.*, together with underapproximation, closed recurrence sets characterize non-termination).

Theorem 3.3 (An Open Recurrence Set always contains a Closed Recurrence Set)**.** *There exists an open recurrence set $\mathcal{G}$ for (S, R, I, F)* iff *there exist an underapproximation (S, R', I', F') of (S, R, I, F) and $\mathcal{G}' \subseteq \mathcal{G}$ such that $\mathcal{G}'$ is a closed recurrence set for (S, R', I', F').*

Proof. "⇐": Any closed recurrence set $\mathcal{G}'$ for (S, R', I', F') with $I' \subseteq I$, $F \subseteq F'$ and $R' \subseteq R$ is also an open recurrence set for (S, R, I, F).

"⇒": Suppose $\mathcal{G}$ is an open recurrence set for (S, R, I, F). Thus there must exist an infinite R-path π in $\mathcal{G}$ with

$$\pi \triangleq s_0 \xrightarrow{R} s_1 \xrightarrow{R} s_2 \xrightarrow{R} \dots$$

and $s_0 \in I$ such that $\{s_i \mid i \geq 0\} \subseteq \mathcal{G}$. We now set $\mathcal{G}' = \{s_i \mid i \geq 0\}$, $I' = \{s_0\}$, and $R' = \{(s_i, s_{i+1}) \mid (s_i, s_{i+1}) \in \pi\}$ and $F' = \{\, s_i \,|\, s_i \in S \wedge \neg\exists\, s_j \in S.\, R'(s_i, s_j)\}$. Thus, $I' \subseteq I$, $R' \subseteq R$ and $F \subseteq F'$ hold by construction, making (S, R', I', F') an underapproximation of (S, R, I, F) and $\mathcal{G}' \subseteq \mathcal{G}$ is a closed recurrence set for (S, R', I', F'). □

Preconditions

When computing preconditions of `assume` statements we borrow from Calcagno *et al.* [CDOY11]: $pre(\texttt{assume}(Q), P) \triangleq P \wedge Q$, called the "assume as assert trick". This lets us interpret `assume` statements (often from conditional branches) in a way that allows us to determine in a precondition the states from which an error location can be reached in a safety counterexample path. The reason is that we want to find out from which states an execution actually makes it to the error location, not states from which we only "get stuck" at an `assume` statement. For assignment statements we will use the standard weakest precondition [Dij76].

Example 3.1. Note that the weakest precondition of an assignment with nondeterminism is a little subtle. Let i := `nondet()`; `assume(true)`; be the `nondet` statement under consideration. The weakest precondition for the postcondition $(\mathsf{i} < \mathsf{j})$ is `false` (equivalent to $\forall \mathsf{i}.\ \mathsf{i} < \mathsf{j}$). However the weakest precondition for the postcondition $(\mathsf{i} < \mathsf{j} \vee \mathsf{k} > 0)$ is $(\mathsf{k} > 0)$.

3.3 Our Procedure

Our non-termination proving procedure PROVER-SAFETY is detailed in Figure 3.3. Its input is a program P given by its CFG, and a loop to be considered for non-termination. To prove non-termination of the entire program P we need to find only one non-terminating loop L. This can be done in parallel. Alternatively, the procedure can

PROVER-SAFETY (CFG P, Loop L)
  $h :=$ header node of L
  $e :=$ exit node of L in P
  $P' :=$ UNDERAPPROXIMATE(P, e)
  $L' :=$ refined loop L in P'
  if $\neg$ REACHABLE(P', h) **then**
    return Unknown, $\bot$
  fi
  $\Pi := \{\pi \mid \pi$ feasible path to h in $P'\}$
  for all $\pi \in \Pi$ **do**
    $P' := \pi :: L'$ // concatenation
    if VALIDATE(P') **then**
      return Non-Terminating, P'
    fi
  done
  return Unknown, $\bot$

UNDERAPPROXIMATE (CFG P, Node e)
  $\kappa :=$ [ ]
  while REACHABLE(P, e) **do**
    $\pi :=$ feasible path to e in P
    $\kappa := \pi :: \kappa$
    $P :=$ REFINE(P, π)
    if the n most recent paths in κ
      are repeating **then**
      $P :=$ STRENGTHEN$(P,$ FIRST$(\kappa))$
    fi
  done
  return P

REFINE (CFG P, Path π)
  $(\ell_0, \ell_1, T_0)(\ell_1, \ell_2, T_1) \ldots (\ell_{n-1}, \ell_n, T_{n-1}) := \pi$
  Calculate WPs $\psi_1, \psi_2 \ldots \psi_{n-1}$ along π
    so $\{\psi_1\}T_1\{\psi_2\}T_2 \ldots \{\psi_{n-1}\}T_{n-1}\{\texttt{true}\}$
    are valid Hoare-triples.
  Find p s.t. $\psi_p \neq \texttt{false} \wedge \forall q < p.\ \psi_q = \texttt{false}$
  $P := P|_{(T_{p-1}, \neg\psi_p)}$
  return P

STRENGTHEN (CFG P, Path π)
  $(\ell_0, \ell_1, T_0)(\ell_1, \ell_2, T_1) \ldots (\ell_{n-1}, \ell_n, T_{n-1}) := \pi$
  Calculate WPs $\psi_1, \psi_2 \ldots \psi_{n-1}$ along π
    so $\{\psi_1\}T_1\{\psi_2\}T_2 \ldots \{\psi_{n-1}\}T_{n-1}\{\texttt{true}\}$
    are valid Hoare-triples.
  Find p s.t. $\psi_p \neq \texttt{false} \wedge \forall q < p.\ \psi_q = \texttt{false}$
  $W := \{v \mid v$ gets updated in subpath
    $(\ell_p, \ell_{p+1}, T_p) \ldots (\ell_{n-1}, \ell_n, T_{n-1})\}$
  $\rho_p :=$ QE$(\exists W.\ \psi_p)$
  $P := P|_{(T_{p-1}, \neg\rho_p)}$
  return P

VALIDATE (CFG P')
  $L' :=$ the outermost loop in P'
  $\mathbb{M} := \{\ell \mid \ell$ is `nondet` assignment node in $L'\}$
  for all $\ell \in \mathbb{M}$ **do**
    Calculate invariant inv_ℓ at node ℓ in P'
    let `nondet` statement at ℓ be
      $\mathsf{v} :=$ `nondet()`; `assume`(φ);
    if $inv_\ell \rightarrow \exists \mathsf{v}.\ \varphi$ is not valid **then**
      return `false`
    fi
  done
  return `true`

Figure 3.3: Procedure PROVER-SAFETY for underapproximation to synthesize a reachable non-terminating loop. To prove non-termination of P, PROVER-SAFETY should be run on all loops L.

be implemented sequentially, but then timeouts are advisable in PROVER-SAFETY, as the procedure might diverge and cause another loop to not be considered.

The subprocedure UNDERAPPROXIMATE performs the search for an underapproximation such that we can prove the loop is never exited. While the loop exit is still REACHABLE (*a.k.a.* "unsafe"), we use the subprocedure REFINE to examine paths returned from an off-the-shelf safety prover. Here the notation $P|_{(T_i,\varphi)}$ denotes P with an additional `assume`(φ) added to the transition T_i. From the postcondition `true` (used to indicate success in reaching the loop exit), we use a backwards precondition analysis to find out which program states will inevitably end up in the loop exit. We continue this precondition calculation until either we have reached the beginning of

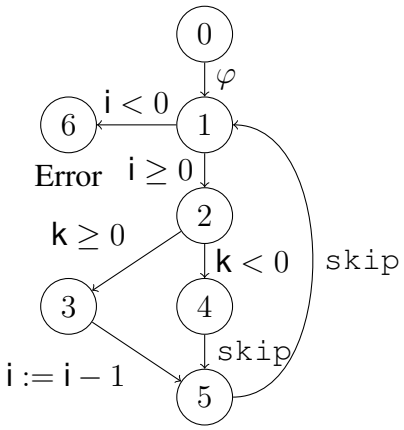


Figure 3.4: Program with Repeating Counterexamples

the path or until just before we have reached a non-deterministic assignment that leads to the precondition `false`. We then negate this condition as our underapproximating refinement and strengthen the previous transition with this refinement. Thus depending on where our precondition analysis ends, we either refine the program's precondition or the assume statement associated with a non-deterministic assignment.

In some cases our refinement is too weak, leading to divergence. The difficulty is that in cases the same loop path will be considered repeatedly, but at each instance the loop will be unrolled for an additional iteration. To avoid this problem we impose a limit n for the number of paths that go along the same locations (possibly with more and more repetitions). We call such paths *repeating*. If we reach this limit, we use the subprocedure STRENGTHEN to strengthen the precondition, inspired by a heuristic by Cook and Koskinen [CK13]. Here we again calculate a precondition, but when we have found ψ_p, we quantify out all the variables that are written to after ψ_p and apply quantifier elimination (QE) to get ρ_p. We then refine with $\neg\rho_p$. This leads to a more aggressive pruning of the transition relation. This heuristic can lead to additional incompleteness.

Example 3.2. Consider the instrumented program in Figure 3.4. Suppose we have initially $\varphi \triangleq \mathsf{i} \geq 0$. We might get $\text{cex}_1 : 0 \to 1 \to 2 \to 3 \to 5 \to 1 \to 6$ as a first counterexample. The REFINE procedure finds the weakest precondition $\mathsf{k} \geq 0 \wedge \mathsf{i} = 0$ at location 1. Adding its negation to φ and simplifying the formula gives us $\varphi \triangleq (\mathsf{i} \geq 0) \wedge (\mathsf{k} < 0 \vee \mathsf{i} \geq 1)$. Now we may get $\text{cex}_2 : 0 \to 1 \to 2 \to 3 \to 5 \to 1 \to 2 \to 3 \to 5 \to 1 \to 6$ as next counterexample, and REFINE updates

$\varphi \triangleq (\mathsf{i} \geq 0) \wedge (\mathsf{k} < 0 \ \vee \ \mathsf{i} \geq 2)$. Now we may get $\text{cex}_3 : 0 \to 1 \to 2 \to 3 \to 5 \to 1 \to 2 \to 3 \to 5 \to 1 \to 2 \to 3 \to 5 \to 1 \to 6$ as next counterexample. Note that cex_1, cex_2, cex_3 are repeating counterexamples and if we just use the REFINE procedure, UNDERAPPROXIMATE gets stuck in a sequence of infinite counterexamples. Now STRENGTHEN identifies the repeating counterexamples, considers cex_1 and calculates the weakest precondition $\psi_1 \triangleq \mathsf{k} \geq 0 \wedge \mathsf{i} = 0$. It then existentially quantifies out variable i as it gets modified later along cex_1. We get $\exists \mathsf{i}.\ \mathsf{k} \geq 0 \wedge \mathsf{i} = 0$, and quantifier elimination yields $\rho_1 \triangleq \mathsf{k} \geq 0$. Clearly ψ_1 entails ρ_1. Adding $\neg\rho_1$ to φ and simplifying the formula we get $\varphi \triangleq \mathsf{i} \geq 0 \wedge \mathsf{k} < 0$. Now all repeating counterexamples are eliminated, the program is safe, and we have obtained a closed recurrence set witnessing non-termination of the original program.

In the UNDERAPPROXIMATE procedure, once there are no further counterexamples to safety of P, we know that in P the loop exit is not reachable. The procedure returns the final underapproximation (denoted by P') that is safe.

When UNDERAPPROXIMATE returns to PROVER-SAFETY, we check if in P' the original loop L after refinements has a closed recurrence set. We refer to the refined loop as L'. In order to check the existence of a closed recurrence set, we first need to ensure that L' is reachable in P' even after the refinements. We again pose this problem as a safety/reachability problem. This time we mark the header node of L' as an error location in P' and hope that P' is unsafe. If P' is safe then clearly we have failed to prove non-termination and we report the result as unknown. If P' is unsafe, then the counterexample to its safety is a path to the header of L'. We enumerate all such paths to the header of L' in a set Π (generated lazily in our implementation). For each such path $\pi \in \Pi$ we then create a simplified CFG P' by concatenating π to L', thus eliminating other paths to L'.

At this point, we are sure that the header of L' is reachable and there is no path that can reach the exit location of L'. However refinements in UNDERAPPROXIMATE may have restricted the `nondet` statements inside L' by strengthening the `assume` statements associated with them. Thus a reachable state at the non-deterministic assignment node may not have a successor along its outgoing edge. This would bring our alleged infinite execution to a halt. The safety checker cannot detect this since then the path just gets blocked at this node, and the error location at the exit of L' cannot be reached.

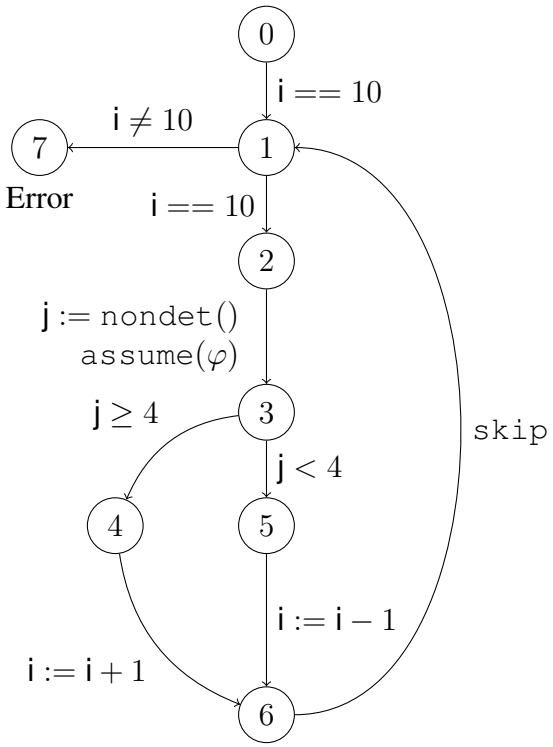


Figure 3.5: Program showing why we need VALIDATE procedure

Example 3.3. Consider the instrumented program in Figure 3.5. Suppose initially $\varphi \triangleq \texttt{true}$. The original program (without instrumentation) is clearly terminating. Our algorithm might give $\text{cex}_1 : 0 \to 1 \to 2 \to 3 \to 4 \to 6 \to 1 \to 7$ as the first counterexample. The `nondet` statement at node 2 gets restricted by updating $\varphi \triangleq (\mathsf{j} \leq 3 \vee \mathsf{i} == 9)$. Now we might get $\text{cex}_2 : 0 \to 1 \to 2 \to 3 \to 5 \to 6 \to 1 \to 7$ as the next counterexample. Our algorithm restricts the `nondet` statement at node 2 by updating $\varphi \triangleq (\mathsf{j} \leq 3 \vee \mathsf{i} == 9) \wedge (\mathsf{j} \geq 4 \vee \mathsf{i} == 11)$.

However now there are no further counterexamples, and the safety checker returns safe. The state $s \vDash \mathsf{i} = 10$ at node 2 has no successor along the outgoing edge as there is no way to satisfy the condition φ and the execution is halted, so it would be unsound to report the result as Non-terminating.

Note that at first it may appear that adding another outgoing edge to node 2 with $\mathsf{j} := \texttt{nondet}(); \texttt{assume}(\neg\varphi);$ and marking the next node as an error node would help us catch the halted execution. However the problem is that this would discover again all of the previously eliminated counterexamples as well. Thus we need a special check by the VALIDATE procedure, which we describe next.

VALIDATE takes as input the final underapproximation P'. It first calculates a location invariant at every non-deterministic assignment node inside the outermost loop L' in P'. Let ℓ be a nondet. assignment node with: v := `nondet()`; `assume`(φ); Let *inv* be a location invariant at ℓ. VALIDATE then checks if (3.4) is valid.

$$inv \rightarrow \exists \mathsf{v}.\, \varphi \tag{3.4}$$

This formula checks if for all reachable states at ℓ, a choice can be made for the `nondet` assignment obeying φ (and thus Condition (3.2) holds). Our implementation uses APRON's [JM09] octagon abstract domain [Min06] to calculate this invariant. VALIDATE returns `true` *iff* (3.4) holds for all `nondet` statements in L'.

Example 3.4. Consider the program in Figure 3.2(f). Using a standard invariant generator we calculate the invariant $\mathsf{i} \geq 0$ before line 4. Substituting in (3.4) we get, $\mathsf{i} \geq 0 \rightarrow \exists \mathsf{i}'.\, \mathsf{i}' \geq 0$. Clearly the formula is valid. Note that in most of the cases even the weakest invariant `true` can be sufficient to prove validity of (3.4). In this example as well we can easily prove that `true` $\rightarrow \exists \mathsf{i}'.\, \mathsf{i}' \geq 0$ is valid.

Moreover, consider the program in Figure 3.5. Suppose $\varphi \triangleq (\mathsf{j} \leq 3 \vee \mathsf{i} == 9) \wedge (\mathsf{j} \geq 4 \vee \mathsf{i} == 11)$. Using an invariant generator, we obtain the location invariant $\mathsf{i} = 10$ at location 2. Then (3.4) becomes $\mathsf{i} = 10 \rightarrow \exists \mathsf{j}.\, (\mathsf{j} \leq 3 \vee \mathsf{i} = 9) \wedge (\mathsf{j} \geq 4 \vee \mathsf{i} = 11)$. Clearly the formula is not valid. In this case VALIDATE returns `false`.

If VALIDATE returns `true`, we are sure that every reachable state at the non-deterministic assignment node in L' has a successor along the edge. At this point, we report non-termination and return the final underapproximation P' of P as a proof of non-termination for P: P' is a closed recurrence set.

Note that as invariants are overapproximations, we may report unknown in some cases even when the discovered underapproximation actually does have a closed recurrence set. However, the check is essential to retain soundness.

Proof of Correctness

We now provide the proof of correctness of our method. Referring to Definition 2.8, we can represent L' in P' as a transition system $(S_{L'}, R_{L'}, I_{L'}, F_{L'})$. Since we have a

unique path π in P' that goes to L', $I_{L'}$ is the strongest postcondition after execution of π. We first prove the following lemma.

Lemma 3.4 (PROVER-SAFETY Finds Closed Recurrence Sets for Loops)**.** *Let P be a program and L a loop in P. Suppose* PROVER-SAFETY $(P, L) =$ *Non-terminating*, P'. *Then the set of all reachable memory states at the loop header h of L' forms a closed recurrence set for $(S_{L'}, R_{L'}, I_{L'}, F_{L'})$.*

Proof. Let $\Gamma \subseteq M$ represent the set of all reachable memory states at the loop header h of L'. We need to show Conditions (3.1), (3.2), and (3.3) for Γ.

For Condition (3.1) for Γ: As L' is reachable via π (because REACHABLE (P', h) holds), we must have $\exists\sigma.\, I_{L'}(\sigma)$. We have $I_{L'} \subseteq \Gamma$. Thus we have $\exists\sigma. I_{L'}(\sigma) \wedge \Gamma(\sigma)$. Thus we have Condition (3.1) for Γ.

For Condition (3.2) for Γ: Let σ such that $\Gamma(\sigma)$. We need to show that there exists some σ' such that $R_{L'}(\sigma, \sigma')$ holds. As finally the safety check in the UNDERAPPROXIMATE loop succeeds, we know that (h, σ) cannot have a successor along the exit edge of L. (Otherwise the error location would be reachable and the safety checker would catch it.) By construction of deterministic conditional nodes, the program state (h, σ) must have a successor along the guard edge of L'.

Every reachable program state at every deterministic conditional node of L must have a successor along one of its outgoing edges. Similarly, every reachable program state at every deterministic assignment node of L must have a successor along its outgoing edge. As VALIDATE returns `true`, every reachable program state at a non-deterministic assignment node in L' also has a successor along its outgoing edge. This ensures that there must be a loop path π from program state (h, σ) to a program state (h, σ') going through the loop body such that $R_\pi(\sigma, \sigma')$. This ensures that $R_{L'}(\sigma, \sigma')$. Thus we have Condition (3.2) for Γ.

For Condition (3.3) for Γ: Let σ, σ' such that $\Gamma(s) \wedge R_{L'}(\sigma, \sigma')$. Clearly σ' is also a reachable memory state at the loop header of L' and the definition of Γ implies $\Gamma(\sigma')$. Thus we have Condition (3.3) for Γ. □

Theorem 3.5 (Correctness of PROVER-SAFETY for Non-termination)**.** *Let P be a program and L a loop in P. Suppose* PROVER-SAFETY $(P, L) =$ *Non-terminating*, P'. *Then P is non-terminating.*

```
int i;
assume(φ);
if (i == 10) {
  while (i > 0) {
    i := i − 1;
    while (i == 0)
      skip;
  }
  assert(false);
}
```

Figure 3.6: Instrumented program for the program of Figure 1.1

Proof. Let L' be the loop in P'. Let π be the path to L' in P'. Let L be the corresponding loop in P before the refinements.

Note that every refinement in PROVER-SAFETY either restricts the `assume` statement at the initial node $\ell_{\mathcal{I}}$ representing the precondition for P or restricts the `assume` statement associated with a non-deterministic assignment statement. Thus every edge (ℓ, ℓ', T) in P gets refined to the edge (ℓ, ℓ', T') such that $\mathcal{R}_{T'} \subseteq \mathcal{R}_T$.

Now let σ such that $I_{L'}(\sigma)$. Then we must have $I_L(\sigma)$, as the memory state σ must be reachable at the header node of L in P as well. This gives $I_{L'} \subseteq I_L$. Let σ, σ' such that $R_{L'}(\sigma, \sigma')$. We must have $R_L(\sigma, \sigma')$. This gives $R_{L'} \subseteq R_L$ and $F_L \subseteq F_{L'}$.

From Lemma 3.4 we have that the reachable memory states at the loop header of L' form a closed recurrence set for $(S_{L'}, R_{L'}, I_{L'}, F_{L'})$. We have $S_{L'} = S_L$. Now Theorem 3.3 implies the existence of a recurrence set for (S_L, R_L, I_L, F_L). This proves that P is non-terminating. □

3.4 Advantages

We now discuss some key advantages of the PROVER-SAFETY procedure.

Clearly the most significant advantage of our procedure is its support for unbounded non-determinism. Previous tools either did not support or had very little support for non-determinism. Additionally our procedure can handle nested loops easily. That is a part of the beauty of the reduction to safety, as existing safety provers (*e.g.* SLAM, IMPACT, *etc.*) handle nested loops with ease. Note that technically we only need to consider an outermost loop.

```
int i, j;
assume(φ);
while (i > 0) {
  i := i + 1;
  j := 2 ;
  while (j > 0)
    j := j − 1;
}
assert(false);
```

Figure 3.7: Instrumented program for the program of Figure 1.2

Example 3.5. Let's revisit Example 1.1. The instrumented program created by our procedure is shown in Figure 3.6. Here initially we have $\varphi \triangleq \texttt{true}$. Here the outer loop decreases the value of i 10 times and then it is the inner loop that is non-terminating. However, it suffices only to consider the outermost loop for safety as the `assert(false)` at the end of the outer loop is not reachable, but the head of the outer loop is reachable, so that we have proved non-termination.

Example 3.6. Let's revisit Example 1.2. The instrumented program created by our procedure is shown in Figure 3.7. Here initially we have $\varphi \triangleq \texttt{true}$. As a first counterexample, we might get a path $\mathsf{i} \leq 0$, and to eliminate it our procedure updates $\varphi \triangleq \mathsf{i} > 0$. After this refinement `assert(false)` at the end of the outer loop is not reachable, but the head of the outer loop is reachable, so that we have proved non-termination.

Our procedure can also prove aperiodic non-termination arising out of single as well as nested loops.

Example 3.7. Let's revisit Example 1.5. The instrumented program created by our procedure is shown in Figure 3.8. Here initially we have $\varphi \triangleq \texttt{true}$. As a first counterexample to safety we might get a path $\mathsf{j} \geq 0, \mathsf{i} < \mathsf{j}$. To eliminate this path our procedure updates $\varphi \triangleq \mathsf{j} < 0 \vee \mathsf{i} \geq \mathsf{j}$. After this refinement `assert(false)` at the end of the loop is not reachable, but the head of the loop is reachable, so that we have proved non-termination.

Example 3.8. Let's revisit Example 1.6. The instrumented program created by our procedure is shown in Figure 3.9. We have initially $\varphi \triangleq \texttt{true}$.

```
int i, j, k;
assume(φ);
if (j ≥ 0) {
  while (i ≥ j) {
    k := i − j;
    if (k > 0) {
      i−−;
    }
    else {
      i := 2 × i + 1;
      j++;
    }
  }
  assert(false);
}
```

Figure 3.8: Instrumented program for the program of Figure 1.3

```
int j, k;
assume(φ);
while (k ≥ 0) {
  k := k + 1;
  j := k;
  while (j ≥ 1)
    j := j − 1;
}
assert(false);
```

Figure 3.9: Instrumented program for the program of Figure 1.4

As a first counterexample our procedure will find the path: $\mathsf{k} < 0$, resulting in updating $\varphi \triangleq \mathsf{k} \geq 0$. After this refinement `assert(false)` at the end of the outer loop is not reachable, but the head of the outer loop is reachable, so that we have proved non-termination.

Our implementation later explained in Chapter 6 uses a safety prover for non-recursive programs with linear integer arithmetic commands. As our method is mainly based on safety proving, the underlying safety prover determines its applicability to programs with other features. As there are number of safety provers available that can handle different programming language features, our method could be extended to support these features (*e.g.* heap, recursion).

3.5 Limitations

Our procedure can only prove non-termination. On terminating programs the procedure is likely to diverge. Even for a non-terminating loop with several branches or subloops inside, there could still be potentially infinitely many terminating paths reaching the loop exit. Our procedure is likely to diverge in such cases. Note that STRENGTHEN procedure is only a heuristic suggested to mitigate the problem of periodic repeating paths, but may not be useful in other cases (*e.g.* aperiodic repeating paths).

3.6 Summary and Outlook

We have introduced a new method of proving non-termination. The idea is to split the reasoning in two parts: a safety prover is used to prove that a loop in an underapproximation of the original program *never* terminates; meanwhile failed safety proofs are used to calculate the underapproximation.

Our technique is not restricted to linear integer arithmetic: Given suitable tools for safety proving and for precondition inference, in principle our approach is applicable to *any* program setting (note that the STRENGTHEN procedure is just an optimization). For future work, *e.g.* heap programs are a highly promising candidate for non-termination analysis via abduction tools for separation logic [CDOY11].

Chapter 4

Proving Non-termination Using Max-SMT

4.1 Introduction

In this chapter we present a new method for proving non-termination of non-deterministic programs that leverages Max-SMT-based invariant generation [LRR13, LORR13]. Our method analyses each *Strongly Connected SubGraph (SCSG)* of a program's control flow graph and, by means of Max-SMT solving, tries to find a formula at every node of the SCSG that satisfies certain properties. First, the formula has to be a *quasi-invariant*, *i.e.*, it must satisfy the consecution condition of inductive invariants, but not necessarily the initiation condition. Hence, if it holds at the node during execution once, then it continues to hold from then onwards. Second, the formula has to be *edge-closing*, meaning that it forbids taking any of the outgoing edges from that node that exit the SCSG. Now, once we have computed an edge-closing quasi-invariant for every node of the SCSG, if a program state is reached that satisfies one of them, then program execution will remain within the SCSG from then onwards. The existence of such a program state is tested with an off-the-shelf reachability checker. If it succeeds, we have proved non-termination of the original program, and the edge-closing quasi-invariants of the SCSG and the trace given by the reachability checker form the witness of non-termination.

Our approach differs from previous methods in two major ways. First, edge-closing quasi-invariants are more generic properties than non-termination witnesses produced by other methods, and thus are likely to carry more information and be more

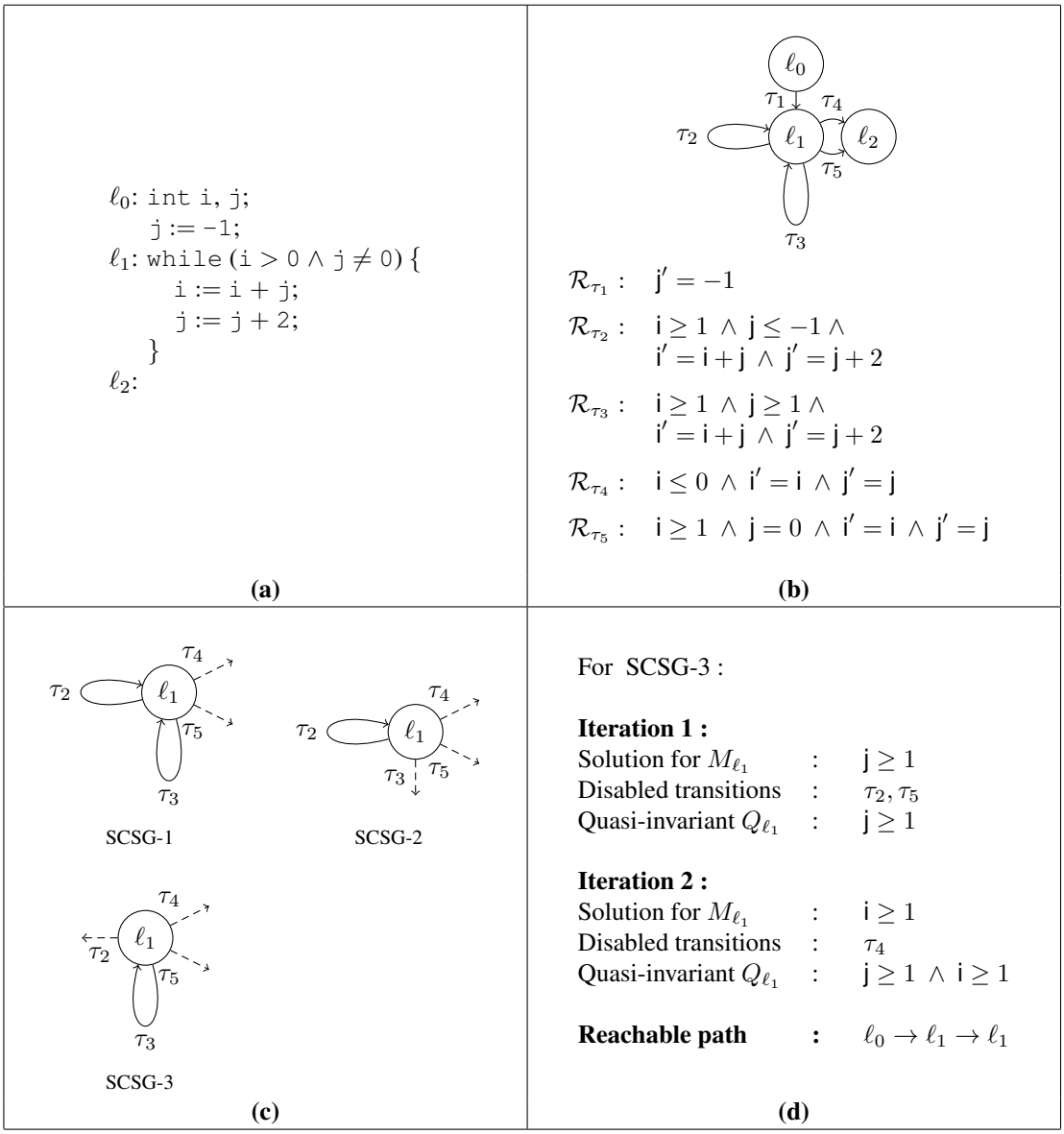


Figure 4.1: Example program (a) together with its corresponding CFG (b), non-trivial SCSGs (c) and non-termination analysis (d)

useful in bug finding. Second, our non-termination witnesses include SCSGs, which are larger structures than, *e.g.*, *lassos*. Note that the number of SCSGs present in any CFG is finite, while the number of lassos is infinite in general. Our method does not suffer from the drawback of divergence of the PROVER-SAFETY procedure described in Section 3.5 as well. Because of these differences, our method is more likely to converge than other methods.

Our technique is based on constraint solving for invariant generation [CSS03] and is goal-directed. Before discussing it formally, we describe it with a simple example.

Consider the program in Figure 4.1(a). The *simplified CFG*[1] for this program is shown in Figure 4.1(b). The edges of the CFG represent the transitions between the locations. For every transition τ, we denote the formula of its transition relation by $\mathcal{R}_\tau(\mathsf{i}, \mathsf{j}, \mathsf{i}', \mathsf{j}')$. The unprimed variables represent the values of the variables before the transition, and the primed ones represent the values after the transition. By $\mathcal{R}_\tau(\mathsf{i}, \mathsf{j})$ we denote the *conditional part of* τ, which only involves the pre-variables. Figure 4.1(c) shows all non-trivial (*i.e.* with at least one edge) SCSGs present in the CFG. For every SCSG, the dashed edges are those that exit the SCSG and hence are not part of it. Note that SCSG-1 is a maximal strongly connected subgraph, and thus is a strongly connected component of the CFG. Notice also that τ_3 is an additional exit edge for SCSG-2, and similarly τ_2 is an exit edge for SCSG-3. The non-termination of this example comes from SCSG-3.

Our approach considers every SCSG of the graph one by one. In every iteration of our method, we try to find a formula at every node of the SCSG under consideration. This formula is originally represented as a template with unknown coefficients. We then form a system of constraints involving the template coefficients in the Max-SMT framework. In a Max-SMT problem, some of the constraints are *hard*, meaning that any solution to the system of constraints must satisfy them, and others are *soft*, which may or may not be satisfied. Soft constraints carry a weight, and the goal of the Max-SMT solver is to find a solution for the hard constraints such that the sum of the weights for the soft constraints violated by the solution is minimized. In our method, essentially the hard constraints encode that the formula should obey the consecution condition, and every soft constraint encodes that the formula will disable an exit edge. A solution to this system of constraints assigns values to template coefficients, thus giving us the required formula at every node.

Consider the analysis of SCSG-3 (refer to Figure 4.1(d)). Note that there is a single node ℓ_1 and a single transition τ_3 in SCSG-3. We denote by $E = \{\tau_2, \tau_4, \tau_5\}$ the set of exit edges for SCSG-3. By $Q_{\ell_1}(\mathsf{i}, \mathsf{j})$ we denote the quasi-invariant at node ℓ_1. Initially $Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \triangleq \texttt{true}$. In the first iteration, for node ℓ_1 we assign a template $M_{\ell_1}(\mathsf{i}, \mathsf{j}) : a.\mathsf{i} + b.\mathsf{j} \leq c$ with template coefficients a, b and c over the integers. After we solve the Max-SMT problem described below, we update Q_{ℓ_1} by conjoining M_{ℓ_1} with

[1] We define simplified CFGs in Section 4.2

the template coefficients instantiated with the model found by the solver.

The Max-SMT problem that we form consists of the following system of hard and soft constraints:

(**Consecution**) $\forall\, \mathsf{i}, \mathsf{j}, \mathsf{i}', \mathsf{j}'.\ M_{\ell_1}(\mathsf{i}, \mathsf{j}) \ \wedge\ Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \ \wedge\ \mathcal{R}_{\tau_3}(\mathsf{i}, \mathsf{j}, \mathsf{i}', \mathsf{j}') \rightarrow M_{\ell_1}(\mathsf{i}', \mathsf{j}')$

(**Edge-Closing**) For all $\tau \in E$: $\forall\, \mathsf{i}, \mathsf{j}.\ M_{\ell_1}(\mathsf{i}, \mathsf{j}) \ \wedge\ Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \rightarrow \neg \mathcal{R}_{\tau}(\mathsf{i}, \mathsf{j})$

The consecution constraint is hard, while the edge-closing constraints are soft (with weight, say, 1). The edge-closing constraint for $\tau \in E$ encodes that, from any state satisfying $M_{\ell_1}(\mathsf{i}, \mathsf{j}) \wedge Q_{\ell_1}(\mathsf{i}, \mathsf{j})$, the transition τ is disabled and cannot be executed.

In the first iteration, a solution for M_{ℓ_1} gives us the formula $\mathsf{j} \geq 1$. This formula satisfies the edge-closing constraints for τ_2 and τ_5. We conjoin this formula to Q_{ℓ_1}, updating it to $Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \triangleq \mathsf{j} \geq 1$. We also update $E = \{\tau_4\}$ by removing τ_2 and τ_5, as these edges are now disabled.

In the second iteration, we again consider the same template $M_{\ell_1}(\mathsf{i}, \mathsf{j})$ and try to solve the Max-SMT problem above with updated $Q_{\ell_1}(\mathsf{i}, \mathsf{j})$ and E. This time we get a solution that gives us the formula $\mathsf{i} \geq 1$, which satisfies the edge-closing constraint for τ_4. We again update $Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \triangleq \mathsf{j} \geq 1 \ \wedge\ \mathsf{i} \geq 1$ by conjoining this new formula. We update $E = \emptyset$ by removing the disabled edge τ_4. Now all exit edges have been disabled, and thus the quasi-invariant $Q_{\ell_1}(\mathsf{i}, \mathsf{j})$ is edge-closing.

In the final step of our method, we use a reachability checker to determine if some state satisfying $Q_{\ell_1}(\mathsf{i}, \mathsf{j})$ at location ℓ_1 is reachable. This test succeeds, and a path $\ell_0 \rightarrow \ell_1 \rightarrow \ell_1$ is obtained. Notice that the path goes through the loop once before we reach the required state. At this point, we have proved non-termination of the original program.

4.2 Preliminaries

In this section we discuss some preliminary concepts which are specific to this chapter.

SMT and Max-SMT

Let $\mathcal{P}$ be a finite set of *propositional variables*. If $p \in \mathcal{P}$, then p and $\neg p$ are *literals*. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals. A *propositional formula* is a conjunction of clauses. The problem of *propositional satisfiability* (abbreviated as SAT) consists of, given a for-

mula, determining whether or not it is *satisfiable*, *i.e.*, if it has a *model*: an assignment of Boolean values to variables that satisfies the formula.

An extension of SAT is the *satisfiability modulo theories (SMT)* problem [BHMW09]: to decide the satisfiability of a given quantifier-free first-order formula with respect to a background theory. In this setting, a model (which we may also refer to as a *solution*) is an assignment of values from the theory to variables that satisfies the formula. Here we will consider the theories of *linear real/integer arithmetic (LRA/LIA)*, where literals are linear inequalities over real and integer variables respectively, and the more general theories of *non-linear real/integer arithmetic (NRA/NIA)*, where literals are polynomial inequalities over real and integer variables, respectively. We will also consider the theory of non-linear integer and real arithmetic *(NIRA)*, where literals are polynomial inequalities over both integer and real variables.

Another generalization of SAT is the *Max-SAT* problem [BHMW09]: it consists of, given a *weighted* formula where each clause has a weight (a positive number or infinity), finding the assignment such that the cost, *i.e.*, the sum of the weights of the falsified clauses, is minimized. Clauses with infinite weight are called *hard*, while the rest are called *soft*. Equivalently, the problem can be seen as finding the model of the hard clauses such that the sum of the weights of the falsified soft clauses is minimized.

Finally, the problem of *Max-SMT* [NO06] merges Max-SAT and SMT, and is defined from SMT analogously to how Max-SAT is derived from SAT. Namely, the *Max-SMT* problem consists of, given a weighted formula, finding an assignment that minimizes the sum of the weights of the falsified clauses in the background theory.

Simplified CFGs

Our technique is applicable to sequential non-deterministic programs with integer variables and commands whose transition relations can be expressed in linear (integer) arithmetic.

In this chapter, the definition of a program differs from Definition 2.2. Note that Definition 2.2 restricts a transition to represent a unique program command. In the new definition a transition can be a composition of several consecutive program commands. This helps us represent a program using a simplified CFG, and a transition can be represented by a formula with several conjuncts, which makes application of Max-SMT

very efficient.

Formally a program is a tuple $(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \mathcal{L}, \mathcal{T}, \Theta)$ where $\overline{\mathsf{v}}$ is a tuple of *program variables*, $\overline{\mathsf{u}}$ is a tuple of *non-deterministic variables*, $\mathcal{L}$ is a set of *locations*, $\mathcal{T}$ is a set of *transitions* and Θ is a formula over $\overline{\mathsf{v}}$ representing the program's precondition. Each transition $\tau \in \mathcal{T}$ is a triple $(\ell, \ell', \mathcal{R})$, where ℓ, $\ell' \in \mathcal{L}$ are the *pre* and *post* locations respectively, and $\mathcal{R}$ is the *transition relation*: a formula over the non-deterministic variables $\overline{\mathsf{u}}$, the program variables $\overline{\mathsf{v}}$ and their primed versions $\overline{\mathsf{v}}'$, which represent the values of the variables after the transition. The transition relation of a non-deterministic assignment of the form $\mathsf{i} := \texttt{nondet}()$, where $\mathsf{i} \in \overline{\mathsf{v}}$, is represented by the formula $\mathsf{i}' = \mathsf{u}_1$, where $\mathsf{u}_1 \in \overline{\mathsf{u}}$ is a fresh non-deterministic variable. Note that u_1 is not a program variable, *i.e.*, $\mathsf{u}_1 \notin \overline{\mathsf{v}}$, and is added only to model the non-deterministic assignment. Thus, without loss of generality on the kind of non-deterministic programs we can model, we will assume that every non-deterministic variable appears in at most one transition relation. A transition that includes a non-deterministic variable in its transition relation is called *non-deterministic* (abbreviated as `nondet`). We represent by $\ell_{\mathcal{I}} \in \mathcal{L}$ and $\ell_{\mathcal{F}} \in \mathcal{L}$, the initial location and the final location of a program respectively.

In what follows we will assume that transition relations are described as conjunctions of linear inequalities over program variables and non-deterministic variables. Given a transition relation $\mathcal{R} = \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}')$, we will use $\mathcal{R}(\overline{\mathsf{v}})$ to denote the *conditional part of* $\mathcal{R}$, *i.e.*, the conjunction of linear inequalities in $\mathcal{R}$ containing only variables in $\overline{\mathsf{v}}$. For a transition system modeling actual programs, the following conditions are true:

$$\text{For } \tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T} : \forall \overline{\mathsf{v}}, \overline{\mathsf{u}}\, \exists \overline{\mathsf{v}}'.\ \mathcal{R}(\overline{\mathsf{v}}) \rightarrow \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}'). \tag{4.1}$$

$$\text{For } \ell \in \mathcal{L} \setminus \{\ell_{\mathcal{F}}\} : \bigvee_{(\ell, \ell', \mathcal{R})} \mathcal{R}(\overline{\mathsf{v}}) \equiv \texttt{true}. \tag{4.2}$$

$$\text{For } \tau_1 = (\ell, \ell_1, \mathcal{R}_1), \tau_2 = (\ell, \ell_2, \mathcal{R}_2) \in \mathcal{T}, \tau_1 \neq \tau_2 : \mathcal{R}_1(\overline{\mathsf{v}}) \wedge \mathcal{R}_2(\overline{\mathsf{v}}) \equiv \texttt{false}. \tag{4.3}$$

Condition (4.1) guarantees that next values for the program variables always exist if the conditional part of the transition holds. Condition (4.2) expresses that, for any location except the final location, at least one of the outgoing transitions from that location can always be executed. Finally, condition (4.3) says that any two different transitions from the same location are mutually exclusive, *i.e.*, conditional branching is always

deterministic.

Example 4.1. This new program definition allows us to create simplified CFGs. Let us consider the program shown in Figure 4.2(a) and its simplified CFG in Figure 4.2(b) that follows the new program definition. Every acyclic path in the original program is represented via a single transition in the CFG using the new program definition. For example the two loop paths are represented using two separate transitions τ_2 and τ_3. Note how the two non-deterministic assignments have been replaced in the CFG by assignments to fresh non-deterministic variables u_1 and u_2. We have $\ell_0 = \ell_{\mathcal{I}}$ and $\ell_2 = \ell_{\mathcal{F}}$. Condition (4.2) is trivially satisfied for ℓ_0, since the conditional part of its outgoing transition relation is `true`. Regarding ℓ_1, clearly the formula $(\mathsf{x} \geq \mathsf{y} \wedge \mathsf{x} \geq 0) \vee (\mathsf{x} \geq \mathsf{y} \wedge \mathsf{x} < 0) \vee (\mathsf{x} < \mathsf{y})$ is a tautology. Condition (4.3) is also easy to check: the conditional parts of $\mathcal{R}_{\tau_2}, \mathcal{R}_{\tau_3}$ and $\mathcal{R}_{\tau_4}$ are pairwise unsatisfiable in conjunction. Finally, condition (4.1) trivially holds since the primed parts of the transition relations consist of equalities whose left-hand side is always a different variable. □

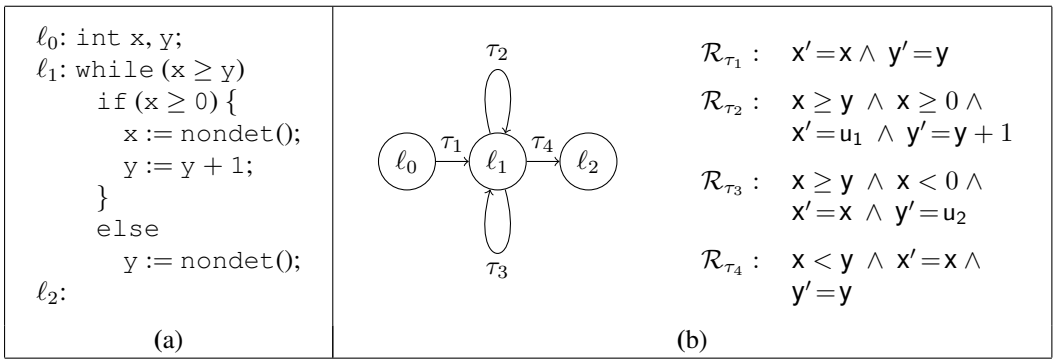


Figure 4.2: Program involving non-deterministic assignments **(a)**, and its simplified CFG **(b)**

The definitions of memory and program states, and computations remain the same as in Chapter 2: now they are only defined over the new program definition.

4.3 Quasi-invariants and Non-termination

Here we will introduce the core concept of this work, that of a *quasi-invariant*: a property such that, if it is satisfied at a location during execution once, then it continues to hold at that location from then onwards. The importance of this notion resides in

the fact that it is a key ingredient in our witnesses of non-termination: if each location of an SCSG can be mapped to a quasi-invariant that is *edge-closing*, *i.e.*, that forbids executing any of the outgoing transitions that leave the SCSG, and the SCSG can be reached at a program state satisfying the corresponding quasi-invariant, then the program is non-terminating (if `nondet` transitions are present, additional properties are required, as will be seen below). A constructive proof of this claim is given at the end of this section.

First of all, let us define basic notation. For a strongly connected subgraph (SCSG) $\mathcal{C}$ of a program's CFG, we denote by $\mathcal{L}^{\mathcal{C}}$ the set of locations of $\mathcal{C}$, and by $\mathcal{T}^{\mathcal{C}}$ the set of edges of $\mathcal{C}$. We define $\mathcal{E}^{\mathcal{C}} \stackrel{def}{=} \{\tau = (\ell, \ell', \mathcal{R}) \mid \ell \in \mathcal{L}^{\mathcal{C}}, \tau \notin \mathcal{T}^{\mathcal{C}}\}$ to be the set of exit edges of $\mathcal{C}$.

Consider a map $\mathcal{Q}$ that assigns a formula $Q_\ell(\overline{\mathsf{v}})$ to each of the locations $\ell \in \mathcal{L}^{\mathcal{C}}$. Consider also a map $\mathcal{U}$ that assigns a formula $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ to each transition $\tau \in \mathcal{T}^{\mathcal{C}}$, which represents the *restriction* that the non-deterministic variables must obey.[2] The map $\mathcal{Q}$ is a *quasi-invariant map* on $\mathcal{C}$ with restriction $\mathcal{U}$ if:

(Consecution)

$$\text{For } \tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^{\mathcal{C}} : \forall \overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}'.\ Q_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}') \wedge U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \rightarrow Q_{\ell'}(\overline{\mathsf{v}}') \quad (4.4)$$

Condition (4.4) says that, whenever a state at $\ell \in \mathcal{L}^{\mathcal{C}}$ satisfying Q_ℓ is reached and a transition from ℓ to ℓ' can be executed, then the resulting state satisfies $Q_{\ell'}$. This condition corresponds to the consecution condition for inductive invariants. Since inductive invariants are additionally required to satisfy initiation conditions [CSS03], we refer to properties satisfying condition (4.4) as quasi-invariants, hence the name for $\mathcal{Q}$.

Example 4.2. In order to explain the roles of $\mathcal{Q}$ and $\mathcal{U}$, consider the program in Figure 4.2. It is easy to see that if $\mathsf{x} \geq \mathsf{y}$ were a quasi-invariant at ℓ_1, the program would be non-terminating (provided ℓ_1 is reachable with a state such that $\mathsf{x} \geq \mathsf{y}$). However, due to the non-deterministic assignments, the property is not a quasi-invariant. On the other hand, if we add the restrictions $U_{\tau_2} := \mathsf{u}_1 \geq \mathsf{x} + 1$ and $U_{\tau_3} := \mathsf{u}_2 \leq \mathsf{y}$, which constrain the non-deterministic choices in the assignments, the quasi-invariant holds

[2] For the sake of presentation, we assume that U_τ is defined for all transitions, whether they are deterministic or not. In the former case, by convention U_τ is `true`.

and non-termination is proved. □

Additionally, our method also needs that $\mathcal{Q}$ and $\mathcal{U}$ are *reachable* and *unblocking*:

$$\textbf{(Reachability)}\ \exists\, \ell \in \mathcal{L}^{\mathcal{C}}.\ \exists\, \sigma \textit{ s.t. } (\ell, \sigma) \text{ is reachable and } \sigma \models Q_\ell(\overline{\mathsf{v}}) \tag{4.5}$$

$$\textbf{(Unblocking)}\ \text{For } \tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^{\mathcal{C}} : \forall \overline{\mathsf{v}} \exists \overline{\mathsf{u}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}) \rightarrow U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \tag{4.6}$$

Condition (4.5) says that there exists a computation reaching a program state (ℓ, σ) such that σ satisfies the quasi-invariant at location ℓ.

As for condition (4.6), consider a memory state σ at some $\ell \in \mathcal{L}^{\mathcal{C}}$ satisfying $Q_\ell(\overline{\mathsf{v}})$. This condition says that, for any transition $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^{\mathcal{C}}$ from ℓ, if σ satisfies the conditional part $\mathcal{R}(\overline{\mathsf{v}})$, then we can always make a choice for the non-deterministic assignment that obeys the restriction $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$.

The last property we require from quasi-invariants is that they are edge-closing. Formally, the quasi-invariant map $\mathcal{Q}$ on $\mathcal{C}$ is *edge-closing* if it satisfies all of the following constraints:

$$\textbf{(Edge-Closing)}\ \text{For } \tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^{\mathcal{C}} : \forall \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \rightarrow \neg \mathcal{R}(\overline{\mathsf{v}}) \tag{4.7}$$

Condition (4.7) says that, from any state at $\ell \in \mathcal{L}^{\mathcal{C}}$ that satisfies $Q_\ell(\overline{\mathsf{v}})$, all the exit transitions are disabled and cannot be executed.

The following is the main result of this section:

Theorem 4.1. *The existence of $\mathcal{Q}$, $\mathcal{U}$ that satisfy (4.4), (4.5), (4.6) and (4.7) for a certain non-trivial SCSG $\mathcal{C}$ of a CFG P imply non-termination of P.*

In order to prove Theorem 4.1, we need the following lemma:

Lemma 4.2. *Let us assume that $\mathcal{Q}$, $\mathcal{U}$ satisfy (4.4), (4.6) and (4.7) for a certain non-trivial SCSG $\mathcal{C}$. Let (ℓ, σ) be a program state such that $\ell \in \mathcal{L}^{\mathcal{C}}$ and $\sigma \models Q_\ell(\overline{\mathsf{v}})$. Then there exists a program state (ℓ', σ') such that $\ell' \in \mathcal{L}^{\mathcal{C}}$, $\sigma' \models Q_{\ell'}(\overline{\mathsf{v}})$ and $(\ell, \sigma) \xrightarrow{\tau} (\ell', \sigma')$ for a certain $\tau \in \mathcal{T}^{\mathcal{C}}$.*

Proof. By condition (4.2) (which is implicitly assumed to hold), there is a transition τ of the form $(\ell, \ell', \mathcal{R})$ for a certain $\ell' \in \mathcal{L}$ such that $\sigma \models \mathcal{R}(\overline{\mathsf{v}})$. Now, by virtue of condition (4.7), since $\sigma \models Q_\ell(\overline{\mathsf{v}})$ we have that $\tau \in \mathcal{T}^{\mathcal{C}}$. Thus, $\ell' \in \mathcal{L}^{\mathcal{C}}$. Moreover,

thanks to condition (4.6) and $\sigma \models Q_\ell(\overline{\mathsf{v}})$ and $\sigma \models \mathcal{R}(\overline{\mathsf{v}})$, we deduce that there exist values ν for the non-deterministic variables $\overline{\mathsf{u}}$ such that $(\sigma, \nu) \models U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$. Further, by condition (4.1) (which is again implicitly assumed), we have that there exists a state σ' such that $(\sigma, \nu, \sigma') \models \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}')$. All in all, by condition (4.4) and the fact that $\sigma \models Q_\ell(\overline{\mathsf{v}})$ and $(\sigma, \nu, \sigma') \models \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}')$ and $(\sigma, \nu) \models U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$, we get that $\sigma' \models Q_{\ell'}(\overline{\mathsf{v}}')$, or equivalently by renaming primed variables in σ' to their unprimed versions, we get $\sigma' \models Q_{\ell'}(\overline{\mathsf{v}})$. So (ℓ', σ') satisfies the required properties. □

Now we are ready to prove Theorem 4.1:

Proof. [of Theorem 4.1] We will construct an infinite computation, which will serve as a witness of non-termination. Thanks to condition (4.5), we know that there exist a location $\ell \in \mathcal{L}^{\mathcal{C}}$ and a memory state σ such that (ℓ, σ) is reachable and $\sigma \models Q_\ell(\overline{\mathsf{v}})$. As (ℓ, σ) is reachable, there is a computation π whose last program state is (ℓ, σ). Now, since $\mathcal{Q}, \mathcal{U}$ satisfy (4.4), (4.6) and (4.7) for $\mathcal{C}$, and $\ell \in \mathcal{L}^{\mathcal{C}}$ and $\sigma \models Q_\ell(\overline{\mathsf{v}})$, we can apply Lemma 4.2 to inductively extend π to an infinite computation of P. □

4.4 Computing Proofs of Non-termination

In this section we explain how proofs of non-termination are effectively computed. As outlined in Section 4.1, first of all we exhaustively enumerate the SCSGs of the CFG. For each SCSG $\mathcal{C}$, our non-termination proving procedure PROVER-MAXSMT, which will be described below, is called. By means of Max-SMT solving, this procedure iteratively computes an unblocking quasi-invariant map $\mathcal{Q}$ and a restriction map $\mathcal{U}$ for $\mathcal{C}$. If the construction is successful and eventually edge-closedness can be achieved, and moreover the quasi-invariants of $\mathcal{C}$ can be reached, then the synthesized $\mathcal{Q}, \mathcal{U}$ satisfy the properties of Theorem 4.1, and therefore the program is guaranteed not to terminate.

In a nutshell, the enumeration of SCSGs considers a strongly connected component (SCC) of the CFG at a time, and then generates all the SCSGs included in that SCC. More precisely, first of all the SCCs are considered according to a topological ordering in the CFG. Then, once an SCC $\mathcal{S}$ is fixed, the SCSGs included in $\mathcal{S}$ are heuristically enumerated starting from $\mathcal{S}$ itself (since taking a strictly smaller subgraph would imply discarding some transitions a priori arbitrarily), then simple cycles in $\mathcal{S}$ (as they are easier to deal with), and then the rest of SCSGs included in $\mathcal{S}$.

PROVER-MAXSMT (SCSG $\mathcal{C}$, CFG P)
  For $\ell \in \mathcal{L}^{\mathcal{C}}$, set $Q_\ell(\overline{\mathsf{v}}) \leftarrow \texttt{true}$
  For $\tau \in \mathcal{T}^{\mathcal{C}}$, set $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \leftarrow \texttt{true}$
  $E^{\mathcal{C}} \leftarrow \mathcal{E}^{\mathcal{C}}$
  while $E^{\mathcal{C}} \neq \emptyset$ **do**
    At $\ell \in \mathcal{L}^{\mathcal{C}}$, assign a template $M_\ell(\overline{\mathsf{v}})$
    At $\tau \in \mathcal{T}^{\mathcal{C}}$, assign a template $N_\ell(\overline{\mathsf{v}}, \overline{\mathsf{u}})$
    Solve Max-SMT problem with
        hard constraints (4.8), (4.9), (4.10) and soft constraints (4.11)
    if no model for hard clauses is found **then return** Unknown, $\perp$ **fi**
    For $\ell \in \mathcal{L}^{\mathcal{C}}$, let $\widehat{M_\ell}(\overline{\mathsf{v}})$ = Solution for $M_\ell(\overline{\mathsf{v}})$
    For $\tau \in \mathcal{T}^{\mathcal{C}}$, let $\widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ = Solution for $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$
    For $\ell \in \mathcal{L}^{\mathcal{C}}$, set $Q_\ell(\overline{\mathsf{v}}) \leftarrow Q_\ell(\overline{\mathsf{v}}) \wedge \widehat{M_\ell}(\overline{\mathsf{v}})$
    For $\tau \in \mathcal{T}^{\mathcal{C}}$ set $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \leftarrow U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge \widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}})$
    Remove from $E^{\mathcal{C}}$ disabled edges
  done
  for all $\ell \in \mathcal{L}^{\mathcal{C}}$ **do**
    if Reachable (ℓ, σ) in P s.t. $\sigma \models Q_\ell(\overline{\mathsf{v}})$ **then**
      let π = reachable path to (ℓ, σ)
      return Non-Terminating, $(\mathcal{Q}, \mathcal{U}, \pi)$
    fi
  done
  return Unknown, $\perp$

Figure 4.3: Procedure PROVER-MAXSMT for proving non-termination of a program P by analyzing SCSG $\mathcal{C}$

Then, once the SCSG $\mathcal{C}$ is fixed, our non-termination proving procedure PROVER-MAXSMT (Figure 4.3) is called. The procedure takes as input an SCSG $\mathcal{C}$ of the program's CFG, and the CFG itself. For every location $\ell \in \mathcal{L}^{\mathcal{C}}$, we initially assign a quasi-invariant $Q_\ell(\overline{\mathsf{v}}) \triangleq \texttt{true}$. Similarly, for every transition $\tau \in \mathcal{T}^{\mathcal{C}}$, we initially assign a restriction $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \triangleq \texttt{true}$. The set $E^{\mathcal{C}}$ keeps track of the exit edges of $\mathcal{C}$ that have not been discarded yet, and hence at the beginning we have $E^{\mathcal{C}} = \mathcal{E}^{\mathcal{C}}$. Then we iterate in a loop in order to strengthen the quasi-invariants and restrictions till $E^{\mathcal{C}} = \emptyset$, that is, all the exit edges of $\mathcal{C}$ are disabled.

In every iteration we assign a template $M_\ell(\overline{\mathsf{v}}) \equiv m_{\ell,0} + \sum_{\mathsf{v} \in \overline{\mathsf{v}}} m_{\ell,\mathsf{v}} \cdot \mathsf{v} \leq 0$ to each $\ell \in \mathcal{L}^{\mathcal{C}}$. We also assign a template $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \equiv n_{\tau,0} + \sum_{\mathsf{v} \in \overline{\mathsf{v}}} n_{\tau,\mathsf{v}} \cdot \mathsf{v} + \sum_{\mathsf{u} \in \overline{\mathsf{u}}} n_{\tau,\mathsf{u}} \cdot \mathsf{u} \leq 0$ to

each $\tau \in \mathcal{T}^{\mathcal{C}}$.[3] Then we form the Max-SMT problem with the following constraints:[4]

- For $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^{\mathcal{C}}$:

$$\forall \overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}'.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}') \wedge U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \rightarrow M_{\ell'}(\overline{\mathsf{v}}') \tag{4.8}$$

- For $\ell \in \mathcal{L}^{\mathcal{C}}$: $\exists \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \bigvee_{\tau=(\ell,\ell',\mathcal{R})\in\mathcal{T}^{\mathcal{C}}} \mathcal{R}(\overline{\mathsf{v}})$ (4.9)

- For $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^{\mathcal{C}}$:

$$\forall \overline{\mathsf{v}} \exists \overline{\mathsf{u}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}) \rightarrow U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \tag{4.10}$$

- For $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^{\mathcal{C}}$: $\forall \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \rightarrow \neg \mathcal{R}(\overline{\mathsf{v}})$ (4.11)

The constraints (4.8), (4.9) and (4.10) are hard, while the constraints (4.11) are soft.

The Max-SMT solver finds a solution $\widehat{M_\ell}(\overline{\mathsf{v}})$ for every $M_\ell(\overline{\mathsf{v}})$ for $\ell \in \mathcal{L}^{\mathcal{C}}$ and a solution $\widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ for every $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ for $\tau \in \mathcal{T}^{\mathcal{C}}$. $\widehat{M_\ell}(\overline{\mathsf{v}})$ and $\widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ are the same as $M_\ell(\overline{\mathsf{v}})$ and $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ respectively, but with the template coefficients instantiated using the model found by the Max-SMT solver. This solution satisfies the hard constraints and as many soft constraints as possible. In other words, it is the best solution for the hard constraints that disables the maximum number of transitions. We then update $Q_\ell(\overline{\mathsf{v}})$ for every $\ell \in \mathcal{L}^{\mathcal{C}}$ by strengthening it with $\widehat{M_\ell}(\overline{\mathsf{v}})$, and update $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ for every $\tau \in \mathcal{T}^{\mathcal{C}}$ by strengthening it with $\widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}})$. We then remove all the disabled transitions from $E^{\mathcal{C}}$ and continue the iterations of the loop with updated $\mathcal{Q}, \mathcal{U}$ and $E^{\mathcal{C}}$. Note that, even if none of the exit edges is disabled in an iteration (*i.e.* no soft constraint is met), the quasi-invariants found in that iteration may be helpful for disabling exit edges later.

When all exit transitions are disabled, we exit the loop with the unblocking edge-closing quasi-invariant map $\mathcal{Q}$ and the restriction map $\mathcal{U}$.

Finally, we check whether there exists a reachable program state (ℓ, σ) such that $\ell \in \mathcal{L}^{\mathcal{C}}$ and $\sigma \models Q_\ell(\overline{\mathsf{v}})$ with an off-the-shelf reachability checker. If this test succeeds, we report non-termination along with $\mathcal{Q}, \mathcal{U}$ and the path π reaching (ℓ, σ) as a witness of non-termination.

The next theorem formally states that PROVER-MAXSMT proves non-termination:

[3] Actually templates $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ are only introduced for `nondet` transitions. To simplify the presentation, we assume that for other transitions, $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ is `true`.

[4] For clarity, the leftmost existential quantifiers over the unknowns of the templates are implicit.

Theorem 4.3. *If procedure* PROVER-MAXSMT *terminates on input SCSG $\mathcal{C}$ and CFG P with* Non-Terminating, *$(\mathcal{Q}, \mathcal{U}, \pi)$, then program P is non-terminating, and $(\mathcal{Q}, \mathcal{U}, \pi)$ allows building an infinite computation of P.*

Proof. Let us prove that, if PROVER-MAXSMT terminates with Non-Terminating, $(\mathcal{Q}, \mathcal{U}, \pi)$, then the conditions of Theorem 4.1, *i.e.*, conditions (4.4), (4.5), (4.6) and (4.7) are met.

First of all, let us prove by induction on the number of iterations of the **while** loop that conditions (4.4) and (4.6) are satisfied, and also that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^{\mathcal{C}} - E^{\mathcal{C}}$,

$$\forall \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \rightarrow \neg \mathcal{R}(\overline{\mathsf{v}}).$$

Before the loop is executed, for all locations $\ell \in \mathcal{L}^{\mathcal{C}}$ we have that $Q_\ell(\overline{\mathsf{v}}) \triangleq \texttt{true}$ and for all $\tau \in \mathcal{T}^{\mathcal{C}}$ we have that $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \triangleq \texttt{true}$. Conditions (4.4) and (4.6) are trivially met. The other remaining condition holds since initially $E^{\mathcal{C}} = \mathcal{E}^{\mathcal{C}}$.

Now let us see that each iteration of the loop preserves the three conditions. Regarding (4.4), by induction hypothesis we have that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^{\mathcal{C}}$,

$$\forall \overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}'.\ Q_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}') \wedge U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \rightarrow Q_{\ell'}(\overline{\mathsf{v}}').$$

Moreover, the solution computed by the Max-SMT solver satisfies constraint (4.8), *i.e.*, has the property that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^{\mathcal{C}}$,

$$\forall \overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}'.\ Q_\ell(\overline{\mathsf{v}}) \wedge \widehat{M_\ell}(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}') \wedge U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge \widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \rightarrow \widehat{M_{\ell'}}(\overline{\mathsf{v}}').$$

Altogether, we have that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^{\mathcal{C}}$,

$$\forall \overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}'.(Q_\ell(\overline{\mathsf{v}}) \wedge \widehat{M_\ell}(\overline{\mathsf{v}})) \wedge \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}') \wedge (U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge \widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}})) \rightarrow (Q_{\ell'}(\overline{\mathsf{v}}') \wedge \widehat{M_{\ell'}}(\overline{\mathsf{v}}')).$$

Hence condition (4.4) is preserved.

As for condition (4.6), the solution computed by the Max-SMT solver satisfies constraint (4.10), *i.e.*, has the property that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^{\mathcal{C}}$,

$$\forall \overline{\mathsf{v}} \exists \overline{\mathsf{u}}.\ (Q_\ell(\overline{\mathsf{v}}) \wedge \widehat{M_\ell}(\overline{\mathsf{v}})) \wedge \mathcal{R}(\overline{\mathsf{v}}) \rightarrow (U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge \widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}})).$$

Thus, condition (4.6) is preserved.

Regarding condition (4.7), note that the transitions $\tau = (\ell, \ell', \mathcal{R}) \in E^{\mathcal{C}}$ that satisfy the soft constraints (4.11), *i.e.*, such that

$$\forall \overline{\mathsf{v}}.\ (Q_\ell(\overline{\mathsf{v}}) \wedge \widehat{M_\ell}(\overline{\mathsf{v}})) \rightarrow \neg \mathcal{R}(\overline{\mathsf{v}})$$

are those removed from $E^{\mathcal{C}}$. Therefore, this preserves the property that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^{\mathcal{C}} - E^{\mathcal{C}}$,

$$\forall \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \rightarrow \neg \mathcal{R}(\overline{\mathsf{v}}).$$

Now, if the **while** loop terminates, it must be the case that $E^{\mathcal{C}} = \emptyset$. Thus, on exit of the loop, condition (4.7) is fulfilled.

Finally, if Non-Terminating, $(\mathcal{Q}, \mathcal{U}, \pi)$ is returned, then there is a location $\ell \in \mathcal{L}^{\mathcal{C}}$ and a state satisfying $\sigma \models Q_\ell(\overline{\mathsf{v}})$ such that configuration (ℓ, σ) is reachable. That is, condition (4.5) is satisfied.

Hence, all conditions of Theorem 4.1 are fulfilled. Therefore, P does not terminate. Moreover, the proof of Theorem 4.1 gives a constructive way of building an infinite computation by means of $\mathcal{Q}, \mathcal{U}$ and π. □

Note that constraint (4.9):

$$\text{For } \ell \in \mathcal{L}^{\mathcal{C}} : \exists \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \bigvee_{\tau=(\ell,\ell',\mathcal{R})\in\mathcal{T}^{\mathcal{C}}} \mathcal{R}(\overline{\mathsf{v}})$$

is not actually used in the proof of Theorem 4.3, and thus is not needed for the correctness of the approach. Its purpose is rather to help PROVER-MAXSMT to avoid getting into dead-ends unnecessarily. Namely, without (4.9) it could be the case that for some location $\ell \in \mathcal{L}^{\mathcal{C}}$, we computed a quasi-invariant that forbids all transitions $\tau \in \mathcal{T}^{\mathcal{C}}$ from ℓ. Since PROVER-MAXSMT only strengthens quasi-invariants and does not backtrack, if this situation were reached the procedure would probably not succeed in proving non-termination.

Now let us describe how constraints are effectively solved. Although it is not necessary, we restrain the coefficients of templates $M_\ell(\overline{\mathsf{v}})$ and $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ to take integer values. This allows us to exploit efficient non-linear solving techniques [BLN+09].

The formula in constraint (4.9) is an existentially quantified formula in NIA and can be directly handled by a Max-SMT solver. The constraints (4.8) and (4.11) are universally quantified over integer variables. Following the same ideas of constraint-based linear invariant generation [CSS03], these constraints are soundly transformed into an existentially quantified formula in NIRA by abstracting program and non-deterministic variables and considering them as reals, and then applying Farkas' Lemma. The Farkas multipliers are reals but the template coefficients are imposed to be integers (*i.e.*, only values from the domain of integers are allowed) and thus this formula is in NIRA. As regards constraint (4.10), the alternation of quantifiers in

$$\forall \overline{\mathsf{v}} \exists \overline{\mathsf{u}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}) \rightarrow U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$$

is dealt with by introducing a template $P_{u,\tau}(\overline{\mathsf{v}}) \equiv p_{u,\tau,0} + \sum_{\mathsf{v} \in \overline{\mathsf{v}}} p_{u,\tau,\mathsf{v}} \cdot \mathsf{v}$ for each $u \in \overline{\mathsf{u}}$ and skolemizing. This yields[5] the formula

$$\forall \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}) \rightarrow U_\tau(\overline{\mathsf{v}}, P_{\overline{\mathsf{u}},\tau}(\overline{\mathsf{v}})) \wedge N_\tau(\overline{\mathsf{v}}, P_{\overline{\mathsf{u}},\tau}(\overline{\mathsf{v}})),$$

which implies constraint (4.10), and to which Farkas' Lemma can be applied as above. Note that, since the Skolem function is not symbolic but an explicit linear function of the program variables, potentially one might lose solutions. Also note that, the coefficients of the templates $P_{u,\tau}(\overline{\mathsf{v}})$ are imposed to be integers. Since program variables have integer type, this guarantees that only integer values are assigned in the non-deterministic assignments of the infinite computation that proves non-termination. Thus the resulting formula is also in NIRA.

Finally, after these transformations, a weighted formula in NIRA containing hard and soft clauses is obtained and the resulting problem is handled by a Max-SMT(NIRA) solver [NO06, BLN+09, LORR14].

There are some other issues about our implementation of the procedure that are worth mentioning. Regarding how the weights of the soft clauses are determined, we follow a heuristic aimed at discarding "difficult" transitions in $\mathcal{E}^{\mathcal{C}}$ as soon as possible. Namely, the edge-closing constraint (4.11) of transition $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^{\mathcal{C}}$ is given a

[5] Again, existential quantifiers over template unknowns are implicit.

weight which is inversely proportional to the number of literals in $\mathcal{R}(\overline{\mathsf{v}})$. Thus, transitions with few literals in their conditional part are associated with large weights, and therefore the Max-SMT solver prefers to discard them over others. The rationale is that for these transitions there may be more states that satisfy the conditional part, and hence they may be more difficult to rule out. Altogether, it is convenient to get rid of them before quasi-invariants become too constrained.

Finally, as regards condition (4.3), our implementation can actually handle transition systems for which this condition does not hold. This may be interesting in situations where, *e.g.*, non-determinism is present in conditional statements, and one does not want to introduce additional variables and locations as was done in Section 4.2 for presentation purposes. The only implication of overriding condition (4.3) is that, in this case, the properties that must be discarded in soft clauses of condition (4.11) are not the transitions leaving the SCSG under consideration, but rather the negation of the transitions staying within the SCSG.

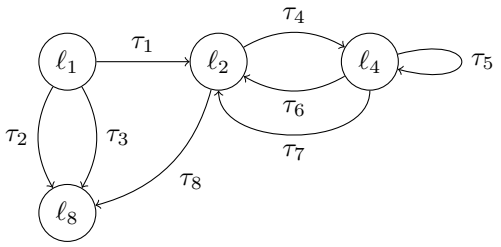
4.5 Advantages

We now discuss some key advantages of the PROVER-MAXSMT procedure.

Similar to the PROVER-SAFETY procedure, PROVER-MAXSMT procedure can handle unbounded non-determinism. As described in Section 3.5 the PROVER-SAFETY procedure needs to eliminate each and every terminating path through a loop and thus it may diverge on many loops. Other techniques like [GHM+08] are also very likely to diverge due to reasons explained in Section 1.2. The PROVER-MAXSMT procedure does not suffer from such drawbacks and is more likely to converge.

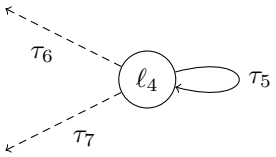
The procedure can handle nested loops easily, as the following examples show.

Example 4.3. Let's revisit Example 1.1. The CFG for this example is shown in Figure 4.4(a). During the analysis of the SCSG shown in Figure 4.4(b), our procedure finds an edge-closing quasi-invariant $\mathsf{i} = 0$ at ℓ_4. This quasi-invariant is reachable and thus we have proved non-termination.



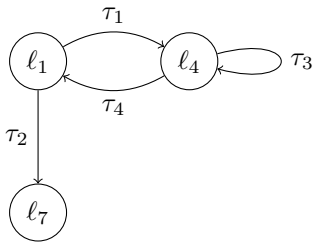
$$
\begin{aligned}
\mathcal{R}_{\tau_1}: &\quad \mathsf{i} = 10 \wedge \mathsf{i}' = \mathsf{i} \\
\mathcal{R}_{\tau_2}: &\quad \mathsf{i} \leq 9 \wedge \mathsf{i}' = \mathsf{i} \\
\mathcal{R}_{\tau_3}: &\quad \mathsf{i} \geq 11 \wedge \mathsf{i}' = \mathsf{i} \\
\mathcal{R}_{\tau_4}: &\quad \mathsf{i} \geq 1 \wedge \mathsf{i}' = \mathsf{i} - 1 \\
\mathcal{R}_{\tau_5}: &\quad \mathsf{i} = 0 \wedge \mathsf{i}' = \mathsf{i} \\
\mathcal{R}_{\tau_6}: &\quad \mathsf{i} \leq -1 \wedge \mathsf{i}' = \mathsf{i} \\
\mathcal{R}_{\tau_7}: &\quad \mathsf{i} \geq 1 \wedge \mathsf{i}' = \mathsf{i} \\
\mathcal{R}_{\tau_8}: &\quad \mathsf{i} \leq 0 \wedge \mathsf{i}' = \mathsf{i}
\end{aligned}
$$

(a)



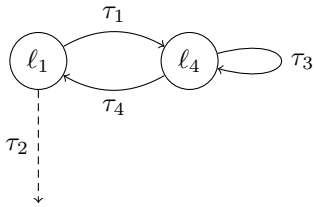
(b)

Figure 4.4: **(a)** Simplified CFG for program of Figure 1.1 and **(b)** the SCSG involved in non-termination



$$
\begin{aligned}
\mathcal{R}_{\tau_1}: &\quad \mathsf{i} \geq 1 \wedge \mathsf{i}' = \mathsf{i} + 1 \wedge \mathsf{j}' = 2 \\
\mathcal{R}_{\tau_2}: &\quad \mathsf{i} \leq 0 \wedge \mathsf{i}' = \mathsf{i} \wedge \mathsf{j}' = \mathsf{j} \\
\mathcal{R}_{\tau_3}: &\quad \mathsf{j} \geq 1 \wedge \mathsf{j}' = \mathsf{j} - 1 \wedge \mathsf{i}' = \mathsf{i} \\
\mathcal{R}_{\tau_4}: &\quad \mathsf{j} \leq 0 \wedge \mathsf{i}' = \mathsf{i} \wedge \mathsf{j}' = \mathsf{j}
\end{aligned}
$$

(a)



(b)

Figure 4.5: **(a)** Simplified CFG for program of Figure 1.2 and **(b)** the SCSG involved in Non-termination

Example 4.4. Let's revisit Example 1.2. The CFG for this example is shown in Figure 4.5(a). During the analysis of the SCC shown in Figure 4.5(b), our procedure finds a quasi-invariant $\mathsf{i} \geq 1$ at both ℓ_1 and ℓ_4. The quasi-invariant at ℓ_1 is edge-closing for the SCC and is also reachable. Thus we have proved non-termination.

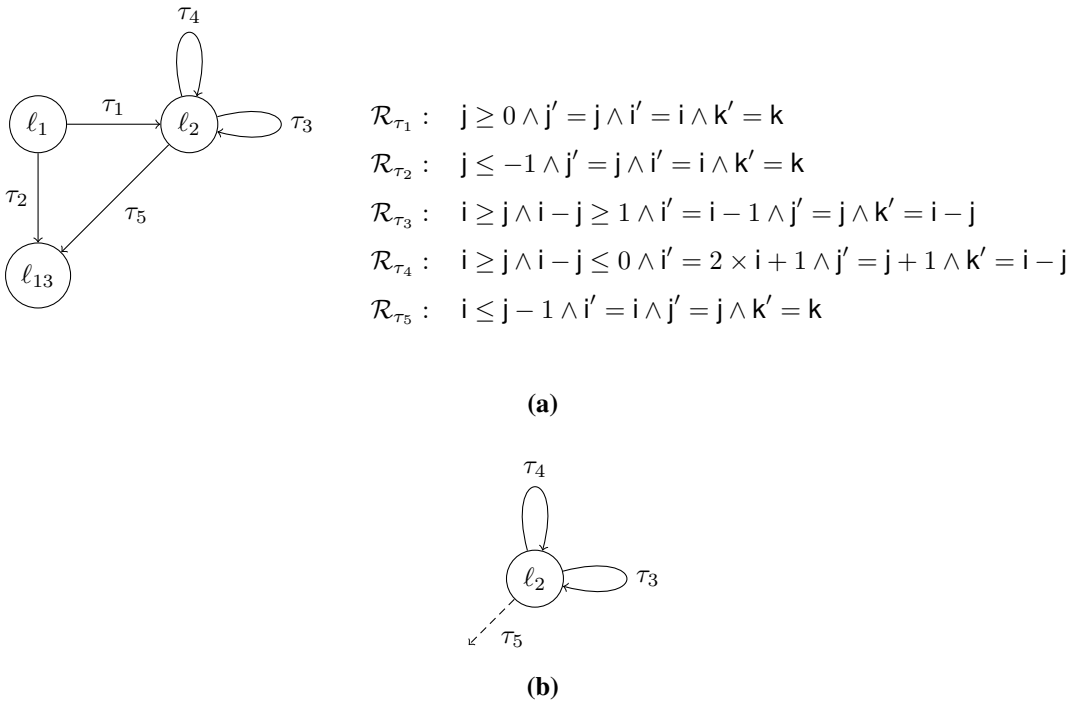


Figure 4.6: **(a)** Simplified CFG for program of Figure 1.3 and **(b)** the SCSG involved in Non-termination

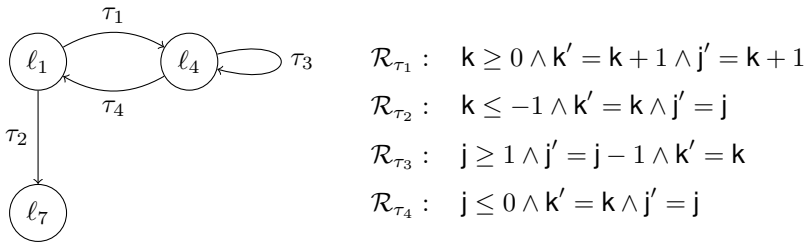
As the following examples show, the PROVER-MAXSMT procedure can also handle aperiodic non-termination arising from single as well as nested loops.

Example 4.5. Let's revisit Example 1.5. The CFG for this example is shown in Figure 4.6(a). During the analysis of the SCC shown in Figure 4.6(b), our procedure finds an edge-closing quasi-invariant $\mathsf{i} \geq \mathsf{j}$ at ℓ_2. Moreover, it is also reachable. Thus we have proved non-termination.

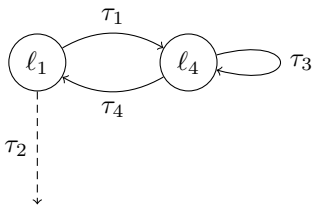
Example 4.6. Let's revisit Example 1.6. The CFG for this example is shown in Figure 4.7(a). During the analysis of the SCC shown in Figure 4.7(b), our procedure finds a quasi-invariant $\mathsf{k} \geq 1$ at both ℓ_1 and ℓ_4. The quasi-invariant at ℓ_1 is edge-closing for the SCC. Moreover, it is also reachable. Thus we have proved non-termination.

4.6 Limitations

The PROVER-MAXSMT is limited to programs with linear integer arithmetic commands. Extending the technique to programs with non-linear arithmetic commands or to programs with other features (*e.g.* heap based commands or real variables) is certainly non-trivial. As described in Section 4.4, the abstraction of integer variables to



(a)



(b)

Figure 4.7: **(a)** Simplified CFG for program of Figure 1.4 and **(b)** the SCSG involved in Non-termination

reals to facilitate the application of Farkas' Lemma, restricting template coefficients to integers and skolemization of non-deterministic variables can potentially lose solutions. This leads to additional incompleteness.

4.7 Summary and Outlook

We have presented a novel Max-SMT-based technique for proving that programs do not terminate. The key notion of the approach is that of a *quasi-invariant*, which is a property such that if it holds at a location during execution once, then it continues to hold at that location from then onwards. The method considers one SCSG of the control flow graph at a time, and thanks to Max-SMT solving, generates a quasi-invariant for each location. Weights of soft constraints guide the solver towards quasi-invariants that are also *edge-closing*, *i.e.*, that forbid any transition exiting the SCSG. If an SCSG with edge-closing quasi-invariants is reachable, then the program is non-terminating. This last check is performed with an off-the-shelf reachability checker.

As regards future research, an important improvement would be to couple the reachability checker with the quasi-invariant generator, so that the invariants synthesized by the former in unsuccessful attempts are reused by the latter when producing quasi-invariants. Another line for future work is to combine the termination technique in [LORR13] and the non-termination technique presented here. Following a simi-

lar approach to [BCF13], if the termination analyzer fails, it can communicate to the non-termination tool the transitions that were proved not to belong to any infinite computation. Conversely, when a failed non-termination analysis ends with an unsuccessful reachability check, one can pass the computed invariants to the termination system, as done in [HLNR10].

Chapter 5

Proving Non-termination with Overapproximation

5.1 Introduction

A program is terminating *iff* its transition relation (when restricted to reachable states) is well-founded. Because every subrelation of a well-founded relation is itself well-founded, if we prove an abstraction that overapproximates the program to be terminating, then we have proved the concrete program terminating. The reverse, unfortunately, is not true: the existence of a non-terminating overapproximating abstraction does not imply that the original concrete program is non-terminating. Thus, when proving non-termination, we currently cannot make use of the many techniques from program analysis that overapproximate programs.

In this chapter we revisit the notion of a *closed recurrence set* and describe a method to prove non-termination that makes use of overapproximation. As proved in Chapter 3, the existence of a closed recurrence set for a program implies that the program does not terminate. Curiously, the existence of a closed recurrence set for an overapproximating abstraction (meeting certain restrictions, which we formalize as *live abstractions*) also implies non-termination of the original concrete program. Thus, when combined with our technique, we can now use overapproximating abstractions when attempting to prove non-termination.

To demonstrate the usefulness of our approach we describe an experimental evaluation where non-linear, non-deterministic, and heap-based programs are proved to be non-terminating using off-the-shelf overapproximating linear abstractions.

```
int i, j, k;
assume (j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
  i := j × k;
  j := j + 1;
  k := k + 1;
ℓ:  skip;
}
```

(a)

```
int i, j, k;
assume (j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0) {
  i := nondet();
  j := j + 1;
  k := k + 1;
ℓ:  assume (i ≥ 1);
}
```

(b)

Figure 5.1: Non-linear program **(a)**, and its linear abstraction **(b)**.

As discussed in detail in the chapter: not all overapproximating abstractions are compatible with our approach. We address this problem by describing the conditions on abstractions that make the abstraction sound for our approach, as the notion of *live abstractions*. Many of the known abstractions indeed meet these conditions. Additionally, closed recurrence sets are not complete, *i.e.* in some cases a closed recurrence set will not exist for non-terminating programs. In these situations our approach can still help in combination with previous techniques to disprove termination (*e.g.* underapproximation) in cases where existing techniques alone could not.

Similar to the tool TNT [GHM+08], our tool can compute the proofs of non-termination only for simple lasso-shaped paths. Furthermore, as done in TNT, when disproving termination of real programs with complex control-flow graphs, we must first search for candidate lassos before applying our approach. Like TNT, our tool also exhaustively searches the program's control flow graph for candidate lassos. Alternatively, candidate lassos can be obtained from a termination prover when it fails to prove termination. Thus our technique can be combined with a termination prover.

5.2 Illustrating Example

Before formally introducing our approach, we first describe the idea informally using an example. Imagine that we want to show non-termination of the toy program in Figure 5.1(a).

We are looking to find initial values for i, j and k from which an infinite run is possible. Indeed, such a run is possible: from the state $(\mathsf{i} = 1, \mathsf{j} = 1, \mathsf{k} = 1)$ the program can perform a sequence of loop iterations via the states $(\mathsf{i} = 1, \mathsf{j} = 2, \mathsf{k} = 2)$,

$(\mathsf{i} = 4, \mathsf{j} = 3, \mathsf{k} = 3)$, $(\mathsf{i} = 9, \mathsf{j} = 4, \mathsf{k} = 4)$, ... leading to an infinite run. This set of states $\mathcal{G} = \{(\mathsf{i} = 1, \mathsf{j} = 1, \mathsf{k} = 1), (\mathsf{i} = 1, \mathsf{j} = 2, \mathsf{k} = 2), (\mathsf{i} = 4, \mathsf{j} = 3, \mathsf{k} = 3), (\mathsf{i} = 9, \mathsf{j} = 4, \mathsf{k} = 4), \ldots\}$ meets the criterion of a recurrence set.

Now the question is, how can we *automatically* find such a proof of non-termination? The difficulty here is the non-linear assignment $\mathsf{i} := \mathsf{j} \times \mathsf{k}$: most automatic formal verification techniques struggle to support non-linear arithmetic in a scalable fashion. An arbitrary overapproximation of this program will not help in this context. The problem is that if we prove non-termination of the overapproximation we still have not proved non-termination of the original concrete program. The reason is that—due to the nature of overapproximation—a non-terminating execution in the overapproximation need not correspond to any execution in the concrete program.

To avoid this problem we can use an overapproximating abstraction of our program such that the abstraction satisfies certain conditions. We call such an abstraction a live abstraction. See Section 5.3. Such an abstraction is shown in Figure 5.1(b). This abstraction uses non-deterministic choice (*i.e.* **nondet**) to abstract away the non-linear command and also uses a linear location invariant at location ℓ from the original program ($\mathsf{i} \geq 1$). Note that in Figure 5.1(b) we do not alter the loop condition from the original program but only overapproximate the transitions that can take place inside the loop. This abstraction is a live abstraction and is thus a safe abstraction for our approach. Later in Section 5.3 we give the necessary conditions for an abstraction to be a live abstraction. Most of the abstractions used in the termination literature satisfy the properties of a live abstraction.

Our approach is based on the following insight: if we can prove existence of a set of states $\mathcal{G}$ at the loop head in the live abstraction meeting the following conditions then we know that both the abstraction and the original concrete program are non-terminating: **a)** $\mathcal{G}$ is nonempty and at least one state in $\mathcal{G}$ is reachable, **b)** every state in $\mathcal{G}$ has at least one transition, and **c)** all transitions from $\mathcal{G}$ in the abstraction only lead to $\mathcal{G}$. In other words, $\mathcal{G}$ meets the conditions of a closed recurrence set defined in Section 3.2. This now allows us to use tools on the overapproximating abstraction rather than the original program to establish non-termination. Here such a set could be given by $\mathcal{G} = \{s \mid s \models \mathsf{i} \geq 1\}$.

```
int i, j, k, m;
assume (j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0 ∧ m ≥ 0) {
  i := j × k;
  j := j + 1;
  k := k + 1;
  m := nondet();
}
```

(a)

```
int i, j, k, m;
assume (j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0 ∧ m ≥ 0) {
  i := j × k;
  j := j + 1;
  k := k + 1;
  m := nondet();
  assume (m ≥ 0);
}
```

(b)

```
int i, j, k, m;
assume (j ≥ 1 ∧ k ≥ 1);
while (i ≥ 0 ∧ m ≥ 0) {
  i := nondet();
  j := j + 1;
  k := k + 1;
  m := nondet();
  assume (m ≥ 0);
  assume (i ≥ 1);
}
```

(c)

Figure 5.2: Non-linear program **(a)**, its underapproximation **(b)**, and the resulting linear abstraction **(c)**.

Combining over- and underapproximation

Sometimes closed recurrence sets are alone not enough: we may still require the use of underapproximation. However, even then, our approach facilitates the *mixture* of over- and underapproximation to make more powerful non-termination proving tools.

Consider the program in Figure 5.2(a). Here it is difficult to find a useful linear overapproximation directly because of the non-deterministic assignment to the variable m. However if an underapproximation of a program is non-terminating, then the original program itself is non-terminating as well. Here we can use known techniques to automatically find an underapproximation that rules out the unwanted transitions. Consider the program in Figure 5.2(b), an underapproximation of the program in Figure 5.2(a) restricting the choice for non-deterministic assignment to the variable m. Using our approach we can now easily find a useful linear overapproximation that is a live abstraction for this program. The program in Figure 5.2(c) is a linear

*over*approximation of the *under*approximation in Figure 5.2(b). Here, we can find a closed recurrence set $\mathcal{G} = \{s \mid s \models \mathsf{i} \geq 1 \wedge \mathsf{m} \geq 0\}$ for the program in Figure 5.2(c), which proves non-termination of the program in Figure 5.2(b), which in turn proves non-termination of the program in Figure 5.2(a). Note that it is unsound to first over-approximate and then underapproximate: as in this example we must first underapproximate and then overapproximate. Also note that for overapproximations we only consider live abstractions.

5.3 Closed recurrence sets and overapproximation

We start with an example which will help us facilitate the discussion in this section.

Example 5.1. Consider the example in Figure 5.1(a). Using Definition 2.8 we can describe the loop and its initial condition as a transition system (S, R, I, F) where any state s is basically a tuple $(\mathsf{i}, \mathsf{j}, \mathsf{k})$ of values of variables and $S = \mathbb{Z}^3$, $R = \{(s, s') \mid s, s' \vDash \mathsf{i} \geq 0 \wedge \mathsf{i}' = \mathsf{j} \times \mathsf{k} \wedge \mathsf{j}' = \mathsf{j} + 1 \wedge \mathsf{k}' = \mathsf{k} + 1\}$, $I = \{s \mid s \vDash \mathsf{j} \geq 1 \wedge \mathsf{k} \geq 1\}$, $F = \{s \mid s \vDash \mathsf{i} < 0\}$.

We now discuss *closed recurrence sets* and their relationship to overapproximation. We first revisit the definition of recurrence sets from Section 2.2. A transition system (S, R, I, F) has a *recurrence set* (or *open recurrence set*) of states $\mathcal{G}$ *iff*

$$\exists s.\mathcal{G}(s) \wedge I(s), \tag{2.1}$$

$$\forall s \exists s'.\mathcal{G}(s) \rightarrow R(s, s') \wedge \mathcal{G}(s'). \tag{2.2}$$

Quantifier alternation as in Condition (2.2) can be a headache for automation. To avoid this problem Gupta *et al.* [GHM+08] restrict the transition relation to deterministic programs only. In this case we can represent the post-state s' using a unique expression in terms of the pre-state s. Thus the existential quantifier can be eliminated by instantiating it with the expression for the post-state.

Example 5.2. For the loop from Figure 5.1(a), we can have a recurrence set $\mathcal{G} = \{(\mathsf{i} = 1, \mathsf{j} = 1, \mathsf{k} = 1), (\mathsf{i} = 1, \mathsf{j} = 2, \mathsf{k} = 2), (\mathsf{i} = 4, \mathsf{j} = 3, \mathsf{k} = 3), (\mathsf{i} = 9, \mathsf{j} = 4, \mathsf{k} = 4), \ldots\}$.

We now revisit the definition of closed recurrence sets from Section 3.2. A set $\mathcal{G}$

is a *closed recurrence set* for a transition system (S, R, I, F) *iff*

$$\exists s.\mathcal{G}(s) \wedge I(s) \tag{3.1}$$

$$\forall s \exists s'.\mathcal{G}(s) \rightarrow R(s, s') \tag{3.2}$$

$$\forall s \forall s'.\mathcal{G}(s) \wedge R(s, s') \rightarrow \mathcal{G}(s') \tag{3.3}$$

This definition of closed recurrence sets has several advantages. Note that in contrast to open recurrence sets, the closed recurrence condition (3.3) is purely universal. Without quantifier alternation, Farkas' Lemma can now be applied directly. This now helps us to incorporate non-deterministic transition systems too. Secondly, as we shall see in Section 5.4, the interaction with overapproximation is improved. The downside is that the condition can be too strong.

The existential quantifier in Condition (3.2) refers only to the (known) transition relation R and, as we shall see in the Section 5.6 on automation, the condition can be easily automated in spite of quantifier alternation when we search for a closed recurrence set $\mathcal{G}$.

Example 5.3. For the loop from Figure 5.1(b), we can have a closed recurrence set $\mathcal{G} = \{s \mid s \vDash \mathsf{i} \geq 1\}$. $\mathcal{G}$ satisfies all the conditions of a closed recurrence set.

5.4 Live abstractions

We now describe generic conditions on abstractions that are sufficient to establish soundness for non-termination proving using our approach, in the form of *live abstractions*.

We assume that an abstraction of $T = (S, R, I, F)$ is a system $T^\alpha = (S^\alpha, R^\alpha, I^\alpha, F^\alpha)$, with a concretion (or meaning) function $[\![\cdot]\!] : S^\alpha \rightarrow \mathcal{P}(S)$.

Definition 5.1 (Live Abstraction)**.** *An abstraction* $T^\alpha = (S^\alpha, R^\alpha, I^\alpha, F^\alpha)$ *is* live *iff*

$$\forall s \forall s' \forall a.\, R(s, s') \wedge s \in [\![a]\!] \rightarrow \exists a'.R^\alpha(a, a') \wedge s' \in [\![a']\!] \quad \text{(Simulation)}$$

$$\forall f \forall g.\, f \in F \wedge f \in [\![g]\!] \rightarrow g \in F^\alpha \quad \text{(Upward Termination)}$$

The *Simulation* (or, 'up simulation') condition is a standard one for overapproximation: it says that any steps we can take in the concrete transition system can be overapproximated in the abstract transition system. The *Upward Termination* condition says that for every final state in the concrete transition system, any corresponding abstract state is also a final state in the abstract transition system. Together *Simulation* and *Upward Termination* imply that for every terminating run in the concrete transition system, also any corresponding run in the abstract transition system is terminating.

The connection of these conditions to disproving termination then is: if there is an initial state a_0 from which all computations in the abstract program are non-terminating and there is an initial state s_0 in the concrete program such that $s_0 \in [\![a_0]\!]$, then all computations in the concrete program starting from s_0 are non-terminating (*i.e.*, for live abstractions, *closed* recurrence carries over from the abstract to the concrete).

Theorem 5.1 (Soundness)**.** Consider a live abstraction $(S^\alpha, R^\alpha, I^\alpha, F^\alpha)$ for a transition system (S, R, I, F). Suppose $\mathcal{G}^\alpha$ is a *closed* recurrence set for $(S^\alpha, R^\alpha, I^\alpha, F^\alpha)$ and for some a_0 we have $\mathcal{G}^\alpha(a_0) \wedge I^\alpha(a_0) \wedge \exists s_0.(\, s_0 \in [\![a_0]\!] \wedge I(s_0)\,)$. Then there also exists a closed recurrence set $\mathcal{G} = \{s \mid \exists a.\, \mathcal{G}^\alpha(a) \wedge s \in [\![a]\!]\}$ for (S, R, I, F).

Proof. We need to prove Conditions (3.1), (3.2), and (3.3) for $\mathcal{G}$.

For Condition (3.1) for $\mathcal{G}$: We have for some a_0, $\mathcal{G}^\alpha(a_0) \wedge I^\alpha(a_0) \wedge \exists s_0.(\, s_0 \in [\![a_0]\!] \wedge I(s_0)\,)$. Thus for such s_0 we have $I(s_0)$ and the definition of $\mathcal{G}$ implies $\mathcal{G}(s_0)$. Thus we have Condition (3.1) for $\mathcal{G}$.

For Condition (3.2) for $\mathcal{G}$: Let s such that $\mathcal{G}(s)$. We now prove that $s \notin F$ by contradiction. Suppose $s \in F$. The definition of $\mathcal{G}$ implies $\exists a.s \in [\![a]\!] \wedge \mathcal{G}^\alpha(a)$. Condition (3.2) for $\mathcal{G}^\alpha$ implies $\exists a'.R^\alpha(a, a')$. However *Upward Termination* implies $a \in F^\alpha$, which implies $\neg\exists a' R^\alpha(a, a')$. Thus we have a contradiction. Thus we must have $s \notin F$. This gives Condition (3.2) for $\mathcal{G}$.

For Condition (3.3) for $\mathcal{G}$: Let s, s' such that $\mathcal{G}(s) \wedge R(s, s')$. The definition of $\mathcal{G}$ implies $\exists a.s \in [\![a]\!] \wedge \mathcal{G}^\alpha(a)$. Moreover, the *Simulation* condition gives $\exists a'.R^\alpha(a, a') \wedge s' \in [\![a']\!]$. Condition (3.3) for $\mathcal{G}^\alpha$ implies $\mathcal{G}^\alpha(a')$. The definition of $\mathcal{G}$ gives $\mathcal{G}(s')$ and thus we have Condition (3.3) for $\mathcal{G}$. □

Note that similar to what many abstractions do, a live abstraction can overapproximate the concrete initial states. For a live abstraction to be useful for proving

non-termination using closed recurrence sets, we only need $a_0 \in S^\alpha$ and $s_0 \in S$ that satisfy the conditions of the soundness theorem.

Example 5.4. Recall Figure 5.1(a) and its abstraction in Figure 5.1(b). We can represent the abstraction as a transition system:

$$I^\alpha = \{a \mid a \vDash \mathsf{j} \geq 1 \wedge \mathsf{k} \geq 1\} \qquad F^\alpha = \{a \mid a \vDash \mathsf{i} < 0\}$$
$$S^\alpha = \mathbb{Z}^3 \qquad R^\alpha = \{(a, a') \mid (a, a') \vDash \mathsf{i} \geq 0 \wedge \mathsf{i}' \geq 1 \wedge \mathsf{j}' = \mathsf{j} + 1 \wedge \mathsf{k}' = \mathsf{k} + 1\}$$

Here a and a' represent states of the abstract transition system. The abstraction contains $\mathsf{i}' \geq 1$ in the transition relation of the loop instead of the non-linear update $\mathsf{i}' = \mathsf{j} \times \mathsf{k}$. Here the abstraction has not changed the state space, the set of initial states and the set of final states, but it has weakened the transition relation of the loop. Note that this abstraction fulfills all criteria for a live abstraction.

Example 5.5. Consider again the examples from Figure 5.1(a) and (b). Here we have the closed recurrence set $\mathcal{G}^\alpha = \{s \mid s \vDash \mathsf{i} \geq 1\}$ for the loop in our abstraction in Figure 5.1(b). This implies existence of a closed recurrence set $\mathcal{G}$ for the loop in the concrete program in Figure 5.1(a) and hence its non-termination.

Example 5.6. To see why we need the *Upward Termination* condition for the abstraction, consider the following transition system (S, R, I, F) and its abstraction $(S^\alpha, R^\alpha, I^\alpha, F^\alpha)$:

$$S = \{s_0, s_1, s_2, s_3\} \quad I = \{s_0\} \quad F = \{s_1\}$$

$$R = \{(s_0, s_1), (s_2, s_3), (s_3, s_0)\}$$

$$S^\alpha = \{\{s_0\}, \{s_1, s_2\}, \{s_3\}\} \quad I^\alpha = \{\{s_0\}\} \quad F^\alpha = \emptyset$$

$$R^\alpha = \{(\{s_0\}, \{s_1, s_2\}), (\{s_1, s_2\}, \{s_3\}), (\{s_3\}, \{s_0\})\}$$

Here an abstract state is a subset of the set of all concrete states, where we have "merged" the states s_1 and s_2 to a single state. The abstraction satisfies the *Simulation* condition but not *Upward Termination* because s_1 is a final state in the con-

crete transition system, but the corresponding abstract state $\{s_1, s_2\}$ is not a final state in the abstract transition system. The abstraction has a closed recurrence set $\{\{s_0\}, \{s_1, s_2\}, \{s_3\}\}$, but the concrete transition system has no recurrence set.

5.5 Classes of Live Abstractions for Automation

As mentioned earlier, in our automation we focus on program fragments of a special shape: lassos.

Definition 5.2 (Lasso)**.** A *lasso* is a program fragment that contains a sequence of commands called a *stem* followed by a *simple loop* with guarded updates. The *guard* of a simple loop is a conjunction of atomic conditions. Formally a lasso L is a transition system $(S, R_{loop}, I_{loop}, F_{loop})$ where S is the set of states in the domain, R_{loop} is the transition relation of the loop, and I_{loop} is the set of initial states for the loop. I_{loop} represents the strongest postcondition after execution of the stem. F_{loop} is a set of final states for the loop such that for every final state there is no transition inside the loop.

Abstracting non-linear commands

We describe the abstraction that our tool uses to abstract non-linear commands present in the lassos. In our abstraction non-linear assignment commands are abstracted, but loop guards are kept unchanged.

Towards the purpose of abstracting assignments we first compute a linear location invariant at the end of the loop (using APRON's [JM09] octagon abstract domain [Min06] in our implementation). We then replace the non-linear update command with a non-deterministic choice and add an assume statement with the invariant at the end of the loop. Instead of octagons, here also dedicated disjunctive analyses for non-linearity (*e.g.* the technique by Alonso *et al.* [ABAG11]) can be used to increase precision of the overapproximation. However, as our experiments show, here we can already get quite far using standard octagons.

Consider the non-linear lasso in Figure 5.1(a) and its linear abstraction in Figure 5.3 that our tool computes. Here, $\mathsf{i}-1 \geq 0 \wedge \mathsf{i}+\mathsf{j}-3 \geq 0 \wedge \mathsf{i}-\mathsf{j}+1 \geq 0 \wedge \mathsf{i}+\mathsf{k}-3 \geq 0$ is the invariant computed at location ℓ of the original lasso from Figure 5.1(a) by the APRON library using the octagon abstract domain.

```
    int i, j, k;
    assume (j ≥ 1 ∧ k ≥ 1);
    while (i ≥ 0 {
      i := nondet();
      j := j + 1;
      k := k + 1;
ℓ:    assume (i − 1 ≥ 0 ∧ i + j − 3 ≥ 0 ∧
        i − j + 1 ≥ 0 ∧ i + k − 3 ≥ 0);
    }
```

Figure 5.3: Linear overapproximation of the program in Figure 5.1(a) computed by our tool using APRON [JM09]

Mapping non-linear assignments to non-deterministic assignments is clearly an overapproximation. This abstraction of assignments satisfies the *Simulation* condition of live abstraction because it adds extra abstract transitions only when a concrete transition (the assignment) is already possible. Since we do not alter loop guards, *Upward Termination* holds as well because all the final states of the original lasso are final states in the abstract lasso too. Clearly this abstraction satisfies the conditions of a live abstraction. Formally for a concrete lasso with a transition system $(S, R_{loop}, I_{loop}, F_{loop})$ our tool computes an abstract lasso with a transition system $(S^\alpha, R^\alpha_{loop}, I^\alpha_{loop}, F^\alpha_{loop})$ where $S^\alpha = S, R_{loop} \subseteq R^\alpha_{loop}, I^\alpha_{loop} = I_{loop}, F^\alpha_{loop} = F_{loop}$ and the concretion function is essentially the identity, *i.e.*, $\forall a \in S^\alpha.\ [\![a]\!] = \{a\}$.

Dealing with non-linear guards

We use a simple trick to get rid of non-linearity in guards. Consider Figure 5.4. We remove non-linearity present in the guards by adding an auxiliary variable v. The rest of the analysis proceeds as before.

This approach yields non-linear commands in the stem of our lassos. The stem commands enter our constraints only existentially (as we will see in Section 5.6). Thus constraint solvers can deal with such constraints efficiently.

Abstracting heap-based commands

Magill *et al.* [MTLT10] propose an overapproximating abstraction from programs operating on the heap to purely arithmetic programs. The abstraction is obtained by instrumenting a memory safety proof for the program. Since in general memory safety only holds under certain preconditions, the user can specify the shape of the heap data

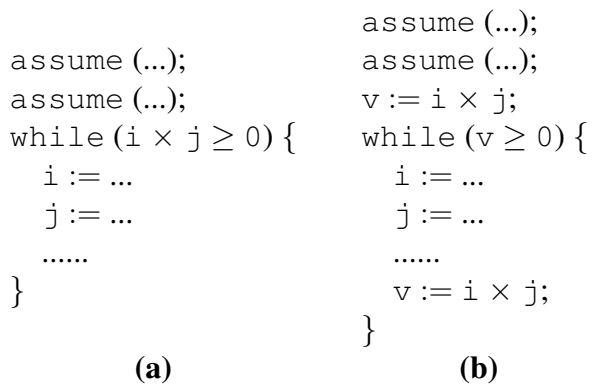


Figure 5.4: Lasso **(a)** with non-linear guards and equivalent lasso **(b)** with auxiliary variable with linear guards

structures by user-defined predicates in separation logic [ORY01]. We can use Magill's tool THOR [MTLT10] to abstract heap-based C programs into linear arithmetic programs operating over the integers. This is exemplified in Figure 5.5. In the arithmetic program the variable k tracks the length of the list segment from p to null, and the other variables are temporaries used in the update of k.

Magill's PhD thesis [Mag10, Def. 29] describes the notion of stuttering simulation and proves (in his Thm. 18) that the abstraction satisfies the properties of stuttering simulation. In stuttering simulation for a transition in the concrete system, the corresponding transition in the abstract system may contain a sequence of steps and vice versa. An abstraction satisfying stuttering simulation obeys standard simulation condition and additionally for stuttering simulation to hold, the *Upward Termination* condition is needed. Thus Magill's abstraction satisfies the properties of a live abstraction and thus is safe for our approach of non-termination proving.

We could also abstract linked-list programs via the results connecting lists and counter automata [BBH+12]. These results are in fact stronger, a bisimulation rather than a simulation, for lists.

Combining over- and underapproximation

As previously mentioned, closed recurrence sets must in some cases be used in conjunction with underapproximation. Here we can use existing techniques for underapproximation in combination with our own. Note that as proved in Theorem 3.3 every non-terminating program also has an underapproximation with a closed recurrence set. Thus closed recurrence sets form a complete method when combined with underap-

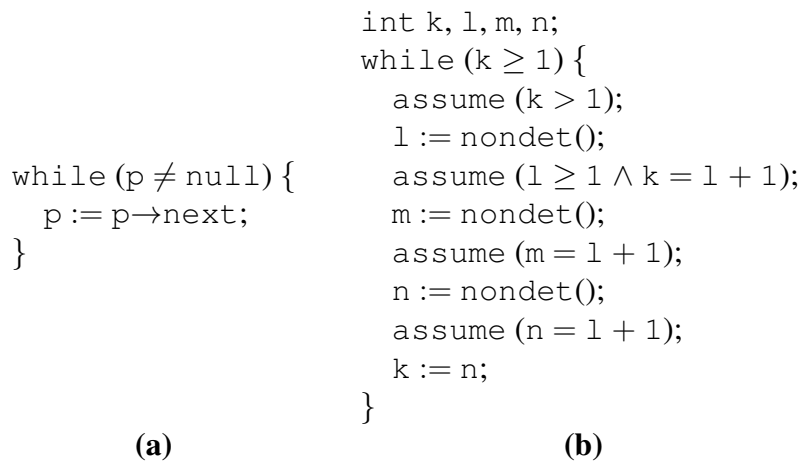


Figure 5.5: Heap-based program **(a)** with precondition that p points to a nonempty cyclic list and linear overapproximation **(b)** computed by THOR [MTLT10]

proximation.

5.6 Finding Closed Recurrence Sets

In the previous section we showed how it is possible to prove non-termination of a program by proving the existence of a closed recurrence set for an abstraction of the program. Here we address the problem of how to *find* a closed recurrence set for the abstracted program, *i.e.*, a program over linear integer arithmetic. We will search for a closed recurrence set $\mathcal{G}$ described by a conjunction of linear inequalities $\boldsymbol{Q}\boldsymbol{x} \leq \boldsymbol{q}$.

We adapt the Farkas-based approach used in TNT to find closed recurrence sets rather than recurrence sets. In our application the restriction to deterministic relations from TNT can be lifted. This is particularly important when working with abstractions of programs, which can introduce non-determinism even when the concrete program is deterministic. It is also essential for treating the heap, because THOR [MTLT10] often introduces non-determinism in the abstractions while handling concrete programs with heap-based commands (*e.g.* `malloc`).

In this section it will be convenient to phrase our discussion in terms of lassos expressed in linear arithmetic, as such lassos are convenient for automation. In the domain of linear arithmetic, a state s is just a vector $\boldsymbol{x}$ that represents the valuation of program variables. A lasso L in linear arithmetic can be expressed as a transition system $(S, R_{loop}(\boldsymbol{x}, \boldsymbol{x'}), I_{loop}(\boldsymbol{x}), F_{loop}(\boldsymbol{x}))$. In terms of programs, $I_{loop}(\boldsymbol{x})$ represents the strongest postcondition of a path leading to the loop body, with precondition 'true'

from which the program starts, and $R_{loop}(\boldsymbol{x}, \boldsymbol{x'})$ is the transition relation corresponding to the composition of a sequence of (possibly non-deterministic) assignment statements in the loop body, guarded by a condition. $F_{loop}(\boldsymbol{x})$ represents the set of final states such that no loop transition can take place from any final state. As we are working in linear arithmetic, we can represent the transition relation of the loop by systems of inequalities

$$R_{loop}(\boldsymbol{x}, \boldsymbol{x'}) \triangleq \boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{g} \wedge \boldsymbol{U}\boldsymbol{x} + \boldsymbol{U'}\boldsymbol{x'} \leq \boldsymbol{u}$$

where $\boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{g}$ describes the guards and $\boldsymbol{U}\boldsymbol{x} + \boldsymbol{U'}\boldsymbol{x'} \leq \boldsymbol{u}$ the updates. Here $\boldsymbol{G}$, $\boldsymbol{U}$ and $\boldsymbol{U'}$ are matrices, $\boldsymbol{g}$ and $\boldsymbol{u}$ are vectors. We make the following assumption:

$$\forall \boldsymbol{x} \exists \boldsymbol{x'}. \boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{g} \rightarrow \boldsymbol{U}\boldsymbol{x} + \boldsymbol{U'}\boldsymbol{x'} \leq \boldsymbol{u}. \tag{5.1}$$

The assumption says that whenever the guards of a lasso can be satisfied we are guaranteed to have a next state given by the updates. This holds in a lasso with a satisfiable transition system when every row in $\boldsymbol{U'}$ contains a non-zero coefficient, which corresponds to an update of the variables.

We are in search of a predicate $\mathcal{G}$ expressed as a system of inequalities using coefficients, *i.e.* $\mathcal{G} \equiv \boldsymbol{Q}\boldsymbol{x} \leq \boldsymbol{q}$, where $\boldsymbol{Q}$ is a matrix and $\boldsymbol{q}$ a vector of existentially quantified variables. The number of rows in $\boldsymbol{Q}$ and $\boldsymbol{q}$ then corresponds to the number of inequalities which we use.

We wish to employ a constraint solver (*e.g.* Z3 [JdM12]) to find the coefficients $\boldsymbol{Q}$ and $\boldsymbol{q}$. A difficulty in doing so is that these conditions contain mixtures of existential and universal quantifiers: $\boldsymbol{Q}$ and $\boldsymbol{q}$ are existentially quantified at the top-level, and both (3.2) and (3.3) use universals. Many constraint solvers struggle to solve problems such as these. Similar to the approach used in Chapter 4, we apply Farkas' lemma to convert the problem into a purely existential one that is easier for existing solvers.

In the remainder of this section we describe a Farkas-based reduction to automate the search for closed recurrence sets. To find a closed recurrence set for $(S, R_{loop}(\boldsymbol{x}, \boldsymbol{x'}), I_{loop}(\boldsymbol{x}), F_{loop}(\boldsymbol{x}))$ we must find $\boldsymbol{Q}$ and $\boldsymbol{q}$ such that the following conditions are satisfied (here we have substituted $\boldsymbol{Q}\boldsymbol{x} \leq \boldsymbol{q}$ for $\mathcal{G}$ in Conditions (3.1), (3.2),

and (3.3)):

$$\exists \boldsymbol{x}.\boldsymbol{Q}\boldsymbol{x} \leq \boldsymbol{q} \wedge I_{loop}(\boldsymbol{x}) \tag{5.2}$$

$$\forall \boldsymbol{x} \exists \boldsymbol{x'}.\boldsymbol{Q}\boldsymbol{x} \leq \boldsymbol{q} \rightarrow R_{loop}(\boldsymbol{x}, \boldsymbol{x'}) \tag{5.3}$$

$$\forall \boldsymbol{x} \forall \boldsymbol{x'}.\boldsymbol{Q}\boldsymbol{x} \leq \boldsymbol{q} \wedge R_{loop}(\boldsymbol{x}, \boldsymbol{x'}) \rightarrow \boldsymbol{Q}\boldsymbol{x'} \leq \boldsymbol{q} \tag{5.4}$$

In order to apply Farkas' lemma we must eliminate the $\forall\exists$ alternation in Condition (5.3).[1] Assumption (5.1) lets us remove the existential quantifier in (5.3),[2] which now becomes:

$$\forall \boldsymbol{x}.\boldsymbol{Q}\boldsymbol{x} \leq \boldsymbol{q} \rightarrow \boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{g} \tag{5.5}$$

Next, although it is not essential, because of (5.5) we can drop $\boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{g}$ from $R_{loop}(\boldsymbol{x}, \boldsymbol{x'})$ in (5.4), thus giving us a simpler constraint to solve:

$$\forall \boldsymbol{x} \forall \boldsymbol{x'}.\boldsymbol{Q}\boldsymbol{x} \leq \boldsymbol{q} \wedge \boldsymbol{U}\boldsymbol{x} + \boldsymbol{U'}\boldsymbol{x'} \leq \boldsymbol{u} \rightarrow \boldsymbol{Q}\boldsymbol{x'} \leq \boldsymbol{q} \tag{5.6}$$

Conditions (5.2), (5.5), and (5.6) are sufficient constraints for finding a closed recurrence set. Furthermore, (5.5) and (5.6) are now in a form which facilitates applications of Farkas' lemma to eliminate the universal quantifiers, and we obtain:

$$\exists \boldsymbol{\Lambda_1} \geq \boldsymbol{0}.\ \boldsymbol{\Lambda_1}\boldsymbol{Q} = \boldsymbol{G} \wedge \boldsymbol{\Lambda_1}\boldsymbol{q} \leq \boldsymbol{g} \tag{5.7}$$

and

$$\exists \boldsymbol{\Lambda_2} \geq \boldsymbol{0}.\ \boldsymbol{\Lambda_2} \begin{pmatrix} \boldsymbol{Q} \\ \boldsymbol{U} \end{pmatrix} = \boldsymbol{0} \wedge \boldsymbol{\Lambda_2} \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{U'} \end{pmatrix} = \boldsymbol{Q} \wedge \boldsymbol{\Lambda_2} \begin{pmatrix} \boldsymbol{q} \\ \boldsymbol{u} \end{pmatrix} \leq \boldsymbol{q} \tag{5.8}$$

Here $\boldsymbol{\Lambda_1}$ and $\boldsymbol{\Lambda_2}$ are matrices. The constraints that we finally generate are (5.2), (5.7), and (5.8). These conditions are readily solved by off-the-shelf constraint solving tools. A satisfying assignment for these constraints gives us values of coefficients in $\boldsymbol{Q}$ and $\boldsymbol{q}$, thus giving us the closed recurrence set.

[1] When Gupta *et al.* [GHM+08] search for recurrence sets, they also need to eliminate the $\forall\exists$ alternation in their constraints for automation. They do so by instantiating the existential variable explicitly with the value of the update. The price for this is that the update must be deterministic. We do not have this restriction.

[2] The statements (5.1) $\wedge$ (5.3) and (5.1) $\wedge$ (5.5) are equivalent.

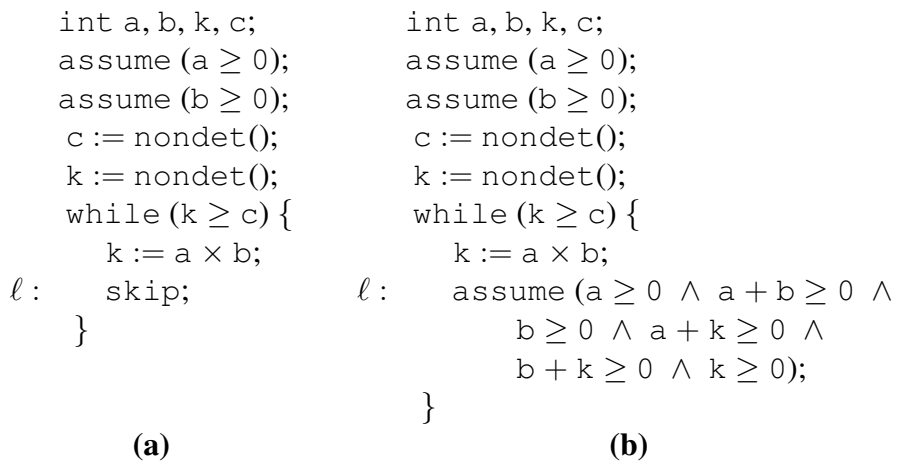


Figure 5.6: Non-linear lasso **(a)** and its live abstraction **(b)**.

Note that if the constraints are unsatisfiable, like Gupta *et al.* [GHM+08] we use $\boldsymbol{Q}$ and $\boldsymbol{q}$ with increasingly many rows (and hence inequalities) in $\boldsymbol{Q}\boldsymbol{x} \leq \boldsymbol{q}$. In this way, we increase the precision of our method further.

Example 5.7. Consider the lasso in Figure 5.6(a) and its live abstraction in Figure 5.6(b). For the live abstraction, we have

$R_{loop}(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{k}, \mathsf{a}', \mathsf{b}', \mathsf{c}', \mathsf{k}') \equiv \mathsf{k} \geq \mathsf{c} \wedge \mathsf{a}' = \mathsf{a} \wedge \mathsf{b}' = \mathsf{b} \wedge \mathsf{c}' = \mathsf{c} \wedge \mathsf{a}' \geq 0 \wedge \mathsf{a}' + \mathsf{b}' \geq 0 \wedge \mathsf{b}' \geq 0 \wedge \mathsf{a}' + \mathsf{k}' \geq 0 \wedge \mathsf{b}' + \mathsf{k}' \geq 0 \wedge \mathsf{k}' \geq 0.$

$I_{loop}(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{k}) \equiv \mathsf{a} \geq 0 \wedge \mathsf{b} \geq 0.$

We get following matrices and vectors.

$$\boldsymbol{x} = \begin{bmatrix} \mathsf{a} \\ \mathsf{b} \\ \mathsf{c} \\ \mathsf{k} \end{bmatrix}, \boldsymbol{G} = \begin{bmatrix} 0 & 0 & 1 & -1 \end{bmatrix}, \boldsymbol{g} = \begin{bmatrix} 0 \end{bmatrix},$$

$$\boldsymbol{U} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad \boldsymbol{U'} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & -1 \\ 0 & -1 & 0 & -1 \\ 0 & 0 & 0 & -1 \end{bmatrix}, \quad \boldsymbol{u} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

We initially try with a single inequality for the closed recurrence set. With this, the system of constraints described above, *i.e.*, the conjunction of (5.2), (5.7), and (5.8) is unsatisfiable, which means that no closed recurrence set exists with a single inequality. We then try with two inequalities for the closed recurrence set. This time we have,

$$\boldsymbol{Q} = \begin{bmatrix} \alpha_1 & \beta_1 & \gamma_1 & \delta_1 \\ \alpha_2 & \beta_2 & \gamma_2 & \delta_2 \end{bmatrix}, \; \boldsymbol{q} = \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix},$$

$$\boldsymbol{\Lambda_1} = \begin{bmatrix} \lambda_1^1 & \lambda_1^2 \end{bmatrix}, \quad \boldsymbol{\Lambda_2} = \begin{bmatrix} \lambda_2^1 & \lambda_2^2 & \dots & \lambda_2^{14} \\ \lambda_3^1 & \lambda_3^2 & \dots & \lambda_3^{14} \end{bmatrix}.$$

This time the system of constraints is satisfiable and we get the following model from the solver.

$$\boldsymbol{Q} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}, \; \boldsymbol{q} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$\boldsymbol{\Lambda_1} = \begin{bmatrix} 0 & 1 \end{bmatrix}, \quad \boldsymbol{\Lambda_2} = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 & 1 & 0 & 1 & 2 & 0 & 1 & 0 & 0 & 1 \end{bmatrix},$$

PROVER-OVERAPPROX (Lasso L)
  L := UNDERAPPROXIMATE (L).
  Create a live abstraction in linear arithmetic L^α of L.
  Solve the problem with constraints (5.2), (5.7), and (5.8).
  if the problem is unsatisfiable **then**
    return Unknown
  fi
  return Non-Terminating

Figure 5.7: Our non-termination proving procedure PROVER-OVERAPPROX

$$\boldsymbol{x} = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}.$$

Note that $\boldsymbol{x}$ in the model above represents the model for constraint (5.2). Thus we get $2.\mathsf{a} + \mathsf{b} + \mathsf{c} \leq 0 \wedge \mathsf{c} - \mathsf{k} \leq 0$ as the closed recurrence set. This proves non-termination of the lasso in Figure 5.6(b), which in turn proves non-termination of the lasso in Figure 5.6(a).

5.7 Our Procedure

Based on the concepts developed in this chapter, the non-termination proving procedure PROVER-OVERAPPROX that our tool implements is given in Figure 5.7. Its input is a lasso L that is to be checked for non-termination. For an entire program P, we need a lasso generator that exhaustively enumerates lassos in P and calls the PROVER-OVERAPPROX procedure on each of them.

Our procedure first uses a suitable underapproximation strategy if necessary. Any valid underapproximation is legitimate in this process. Our tool uses an underapproximation strategy as follows. If a lasso under consideration contains a loop variable with a non-deterministic update that also appears in the loop guard, our tool adds an **assume**-statement at the end of the loop body that enforces the loop guard (as done for variable m in Figure 5.2(b)).

For every lasso L our procedure creates a live abstraction L^α in linear arithmetic. Our tool particularly uses abstractions for heap and non-linear commands described in Section 5.3. We then try to solve a problem with constraints (5.2), (5.7), and (5.8) for the transition system of L^α. If the problem is unsatisfiable, we return Unknown; else

we have found a closed recurrence set for L^α which in turn proves non-termination of L. We provide more details about our tool in Chapter 6.

5.8 Advantages

We now discuss some key advantages of the PROVER-OVERAPPROX procedure.

It is the first method that we know of which soundly uses overapproximation for proving non-termination. The procedure can handle programs with features like non-linear arithmetic and heap-based commands which other methods either do not support or have difficulty dealing with. Although the procedure uses specific underapproximations and live abstractions, technically any valid underapproximations and live abstractions can be used. This highlights the flexibility provided by the procedure and provides new research directions to identify suitable underapproximation strategies and classes of live abstractions.

Comparison with a method based on invariant

Consider the program in Figure 5.1(a). Here $\mathsf{i} \geq 1$ is an invariant at location ℓ. Every reachable state at ℓ must satisfy the invariant. This invariant implies the loop guard $\mathsf{i} \geq 0$. What this means is, for every reachable state at location ℓ, the loop guard always holds. Thus if there exists a state at location ℓ that is reachable, we have proved non-termination. This method for proving non-termination was suggested by Brotherston [Bro15]. The main advantage of this method is that, we do not need overapproximation for proving non-termination.

The PROVER-OVERAPPROX procedure is more powerful than the method described above. For example, consider the program in Figure 5.6(a). Using the octagon abstract domain [Min06], we can compute the invariant $\mathsf{a} \geq 0 \wedge \mathsf{a}+\mathsf{b} \geq 0 \wedge \mathsf{b} \geq 0 \wedge \mathsf{a}+\mathsf{k} \geq 0 \wedge \mathsf{b}+\mathsf{k} \geq 0 \wedge \mathsf{k} \geq 0$ at location ℓ. This invariant does not imply the loop guard $\mathsf{k} \geq \mathsf{c}$. Thus the method described above cannot prove non-termination of this example. However the PROVER-OVERAPPROX procedure can prove non-termination as shown in Example 5.7.

5.9 Limitations

We now discuss some key limitations of the PROVER-OVERAPPROX procedure. Similar to the method of [GHM+08], the procedure can only prove non-termination of lassos

and thus suffers from similar drawbacks. PROVER-OVERAPPROX cannot be useful in detecting aperiodic non-termination. As the number of lassos in a program are infinite the procedure can often diverge. Additionally many of the lassos can in fact be terminating, but the procedure will spend significant amount of time in analysing them. However technically the procedure can be combined with a termination prover where the procedure can be invoked only for lassos on which the termination prover fails to find a termination argument.

5.10 Summary and Outlook

Overapproximation is the workhorse of program analysis. Unfortunately, overapproximation can invalidate conventional techniques for disproving termination. We have introduced the notion of a *live abstraction* to show how overapproximation can *help*, not hinder non-termination proving. The idea is to prove the existence of a *closed recurrence set* rather than simply a recurrence set. This modification in strategy allows us to use off-the-shelf overapproximating abstractions, leading to a new set of methods for disproving termination of real programs.

The restriction to lassos was mainly imposed for the sake of automation, as lassos are very convenient to deal with. However technically any live abstraction is sound for proving non-termination if we aim to find closed recurrence sets. For future work it will be interesting to check if this restriction can be lifted. Additionally, we have only demonstrated simple ways of getting live abstractions. Another research direction can aim at identifying more useful classes of live abstractions.

Chapter 6

Experiments

In this chapter we provide the experimental results and compare the non-termination proving procedures of Chapter 3, Chapter 4 and Chapter 5. These procedures have been implemented in the following tools.

1. T2: The PROVER-SAFETY procedure of Chapter 3 is implemented within the tool T2 [CSZ13, BCF13]. As described previously, although the PROVER-SAFETY procedure could be extended to support other programming language features (*e.g.* heap, recursion), this implementation only considers non-recursive programs with integer variables and linear expressions. As underlying safety-proving backend, we use the interpolating safety prover IMPACT [McM06] implemented in T2. The version of T2 that implements the PROVER-SAFETY procedure for non-termination proving is currently not available in the public domain. However it is expected to be made available in the next source code release of T2.

2. CPPINV: The PROVER-MAXSMT procedure of Chapter 4 is implemented within the tool CPPINV. As a reachability checker it uses CPA [BK11]. This implementation also only considers non-recursive programs with integer variables and linear expressions. The tool is available at the following URL:

   `www.lsi.upc.edu/~albert/cppinv-CAV.tar.gz`

3. ANANT: The PROVER-OVERAPPROX procedure of Chapter 5 is implemented in the tool ANANT. Given a program's CFG, ANANT exhaustively searches for candidate lassos.[1] For every lasso the tool applies the PROVER-OVERAPPROX

[1] ANANT uses the same syntax for transition systems as the termination prover T2 [BCF13]. For heap-based programs in C syntax, the lasso extraction is currently conducted manually.

procedure, using Z3 [dMB08] with the procedure of [JdM12] as the constraint solver.

We make ANANT available for download along with its source code at the following URL:

http://www0.cs.ucl.ac.uk/staff/K.Nimkar/live-abstraction

6.1 Experimental Results on Programs with Linear Integer Arithmetic

In this section we provide experimental results on non-recursive programs with integer variables and linear expressions. We have compared T2, CPPINV and ANANT against following tools.

- TNT [GHM+08], the original TNT tool was not available, and thus we have reimplemented its constraint-based algorithm with Z3 [dMB08] as the SMT backend.
- APROVE [GBE+14], via the Java Bytecode frontend, using the SMT-based non-termination analysis by Brockschmidt *et al.* [BSOG12].
- JULIA [SMP10], which implements an approach via a reduction to constraint logic programming described by Payet and Spoto [PS09].

Like Brockschmidt *et al.* [BSOG12], we were unable to obtain a working version of the tool INVEL [VR08]. Note that in the empirical evaluation by Brockschmidt *et al.* [BSOG12], the APROVE tool (which we have compared against) subsumed INVEL on INVEL's data set.

As a benchmark set, we used a set of 492 benchmarks for termination analysis from a variety of applications also used in prior tool evaluations (*e.g.* Windows device drivers, the APACHE web server, the POSTGRESQL server, integer approximations of numerical programs from a book on numerical recipes [PTVF89], integer approximations of benchmarks from LLBMC [MFS12] and other tool evaluations).

Of these, 81 are known to be non-terminating and 254 terminating. For 157 examples, the termination status is unknown. These examples include a program whose

	(a)			(b)			(c)		
	Nonterm	TO	No Res	Nonterm	TO	No Res	Nonterm	TO	No Res
CPPINV	70	6	5	0	16	238	113	35	9
T2	51	0	30	0	45	209	82	3	72
ANANT	26	7	48	0	58	196	40	15	102
TNT	19	3	59	0	48	206	32	12	113
APROVE	0	61	20	0	142	112	0	139	18
JULIA	3	8	70	0	40	214	0	91	66

Figure 6.1: Evaluation success overview, showing the number of problems solved for each tool. Here **(a)** represents the results for known non-terminating examples, **(b)** is known terminating examples, **(c)** is (previously) unknown examples.

termination would imply the Collatz conjecture, and the remaining examples are too large to render a manual analysis feasible. About 18% of these benchmarks are purely deterministic and rest contain some form of non-determinism. On average a CFG in our test suite has 18.4 nodes (max. 427 nodes) and 2.4 loops (max. 120 loops).

Unfortunately each tool requires a different machine configuration, and thus a direct comparison is difficult. T2 was run on a dualcore Intel Core 2 Duo U9400 (1.4 GHz, 2 GB RAM, Windows 7). CPPINV, ANANT and TNT were run on Intel Core i5-2520M (2.5 GHz, 8 GB RAM, Ubuntu Linux 12.04). We ran APROVE on Intel Core i7-950 (3.07 GHz, 6 GB RAM, Debian Linux 7.2). For JULIA, an unknown cloud-based configuration was used. All tools were run with a timeout of 60s. When a tool returned early with no definite result, we display this outcome in the table in the special column named “No Res”.

We ran three sets of experiments: (a) all the examples previously known to be non-terminating, (b) all the examples previously known to be terminating, and (c) all the examples where no previous results are known. With (a) we assess the efficiency of the procedures, (b) is used to demonstrate the soundness of the procedures, and (c) checks if our procedures scale well on relatively large and complicated examples. The results of the three sets of experiments are given in Figure 6.1, which shows for each tool and for each set (a)–(c) the numbers of benchmarks with non-termination proofs (“Nonterm”), timeouts (“TO”), and no results (“No Res”). (Proofs of *termination*, found by APROVE and JULIA, are also listed as “No Res”.)

Discussion

The poor precision of APROVE & JULIA is mainly due to the non-deterministic updates originally present in many of the benchmarks and also introduced by the (automated) conversion of the benchmarks to Java (the two tools' input syntax). This shows the lack of reliable support of non-determinism in these non-termination tools.

The TNT algorithm requires outright that non-determinism must not occur in the input. Our implementation of TNT softens this requirement slightly: parts of the program with `nondet`-assignments are allowed as long as they are not used during the synthesis of recurrence sets. However this support is still very limited. Moreover TNT does exhaustive enumeration of lassos present in a program and checks each lasso for a witness of non-termination and thus is likely to diverge in this process very often. Thus TNT can only find a few proofs of non-termination.

ANANT does exhaustive enumeration of lassos similar to TNT. However ANANT performs better than TNT due to better support for non-determinism. Note that for a lasso with only linear integer arithmetic the overapproximation computed by ANANT is same as that of the concrete lasso. Thus the real power of overapproximation for finding proofs of non-termination cannot be tested here.

T2 is the second most successful tool and can find many more proofs of non-termination. This shows the usefulness of the safety based approach for proving non-termination presented in Chapter 3. It also indicates significant improvement over previous approaches when input programs involve non-determinism.

CPPINV performs the best and finds the highest number of proofs of non-termination. These results show the usefulness of the Max-SMT-based approach for proving non-termination and indicate significant improvements over previous approaches.

We now compare ANANT, T2 and CPPINV based on the number of successful non-termination proofs they could find. On 15 examples from the benchmark set ANANT succeeded in proving non-termination, but T2 did not. However, T2 could find 82 proofs that ANANT could not. On just one example ANANT succeeded in proving non-termination, but CPPINV did not. However, CPPINV could find 118 proofs that ANANT could not. T2 could find 8 proofs that CPPINV could not. However, CPPINV could find 58 proofs that T2 could not. These results show that no tool subsumes non-

termination proofs of the other tools and suggest that each tool has different power. However in general, CPPINV performs quite consistently.

Note that the latest version of CPPINV that also implements the termination proving procedure described in [LORR13], participated in 2014 Termination Competition and was the winner of "Integer Transition Systems" category. CPPINV could find about 165 non-termination proofs that no other tool could find. These results can be seen at

http://nfa.imn.htwk-leipzig.de/termcomp/competition/20

6.2 Experimental Results on Programs with Non-linear Integer Arithmetic and Heap based Commands

These experiments were conducted to test the usefulness of overapproximation based approach of proving non-termination implemented in ANANT.

Note that among the tools described in this chapter T2, CPPINV and TNT are not applicable, as they do not support programs with non-linear or heap-based commands. Thus we have compared ANANT against APROVE and JULIA with the same settings as in Section 6.1. As a benchmark set, we have gathered 33 example programs containing non-linear, and heap-based commands from various sources. Since non-termination usually indicates a bug, some of our benchmarks implement functions computing factorial, logarithm, *etc.*, with typical programming mistakes that lead to non-termination. The set also includes the non-terminating examples from Berdine *et al.* [BCDO06], in particular the bug in a Windows device driver discussed in this paper. While Berdine *et al.* report that their analysis uncovers this bug by absence of a successful termination proof, we can now go a step further and actually *prove* non-termination of such heap programs.

Figure 6.2 shows the results of our experiments with ANANT, APROVE, and JULIA. We ran ANANT and APROVE on an Intel i7-2640M CPU clocked at 2.8 GHz under Linux. For JULIA, an unknown cloud-based configuration was used. All tools were run with 600 s timeout. As Figure 6.2 shows, ANANT succeeded on 29 of 33 benchmarks, whereas APROVE and JULIA succeeded on only 2 and 4 benchmarks, respectively. This difference is not surprising since overapproximation was thus far not applicable to proving non-termination for non-linear and heap-based programs. In

	ANANT		APROVE		JULIA	
Benchmark	Res	Runtime	Res	Runtime	Res	Runtime
1	✓	0.50 s	×	*timeout*	×	7.01 s
2	✓	0.55 s	×	*timeout*	×	7.80 s
2a	✓	0.82 s	×	*timeout*	×	12.01 s
3	✓	0.56 s	×	*timeout*	×	7.74 s
4	✓	125.66 s	×	*timeout*	×	12.85 s
5	✓	0.45 s	×	18.59 s	×	7.24 s
6	✓	0.48 s	×	235.79 s	✓	7.70 s
7	✓	0.59 s	×	23.51 s	✓	11.83 s
8	✓	0.26 s	×	3.15 s	✓	5.08 s
9	✓	243.00 s	×	5.10 s	✓	6.72 s
10	✓	246.83 s	×	27.42 s	×	11.29 s
11	✓	0.63 s	×	*timeout*	×	8.69 s
12	×	2.35 s	×	*timeout*	×	10.67 s
13	×	1.40 s	×	108.61 s	×	8.54 s
14	✓	121.69 s	×	147.54 s	×	7.33 s
15	✓	131.80 s	×	*timeout*	×	8.45 s
16	✓	57.41 s	×	18.81 s	×	7.07 s
17	✓	0.54 s	×	24.18 s	×	7.06 s
18	×	0.66 s	×	28.03 s	×	6.92 s
19	✓	0.44 s	×	*timeout*	×	7.27 s
20	×	0.74 s	×	*timeout*	×	6.95 s
factorial	✓	0.38 s	×	*timeout*	×	7.57 s
log	✓	0.46 s	×	3.17 s	×	8.59 s
log_by_mul	✓	0.63 s	×	*timeout*	×	7.68 s
lasso_ex1	✓	0.45 s	×	*timeout*	×	7.03 s
lasso_ex2	✓	1.21 s	×	72.25 s	×	8.79 s
lasso_ex3	✓	0.48 s	×	*timeout*	×	7.28 s
nCr_combi	✓	0.70 s	×	10.45 s	×	17.26 s
power	✓	0.43 s	×	*timeout*	×	7.03 s
Create	✓	3.47 s	✓	1.75 s	×	4.94 s
Insert	✓	177.69 s	×	16.86 s	×	7.77 s
Traverse	✓	1.23 s	✓	2.12 s	×	50.28 s
WindowsBug	✓	21.69 s	×	14.46 s	×	50.92 s

Figure 6.2: Results ("Res") and runtimes of ANANT, APROVE, and JULIA on 29 benchmarks with non-linear arithmetic and 4 heap-based benchmarks from Berdine *et al.* [BCDO06]. Here ✓ denotes that the tool proved non-termination, × means that the tool returned without a definite answer, and *timeout* means that the run was terminated externally after 600 s.

contrast, as our experiments show, we can now prove non-termination in many such cases.

It is worth highlighting that *e.g.* on benchmark 9, ANANT took over 4 min to prove non-termination, *vs.* JULIA's <7 s. This difference may partly be due to different machine configurations. However, note that a combined prover for termination and non-termination (like APROVE or JULIA) can discard parts of the program proved terminating and only analyze the rest for non-termination. This can lead to a more focused search for a non-termination proof than ANANT's approach of enumerating arbitrary lassos (whose termination might be easy to prove). Thus, ideally, the procedure implemented in ANANT should be combined with a termination prover.

These results confirm the usefulness of the overapproximation-based approach of proving non-termination described in Chapter 5.

6.3 Summary

The experimental evidence provided in this chapter confirms the usefulness of non-termination proving methods introduced in this thesis. Both PROVER-SAFETY and PROVER-MAXSMT are overwhelmingly successful when compared against other tools on programs with linear integer arithmetic. Additionally the experiments show lack of reliable support for non-determinism in the existing tools for proving non-termination, whereas both PROVER-SAFETY and PROVER-MAXSMT procedures are extremely effective in handling non-determinism. The experiments also highlight lack of support in the existing tools for programs with non-linear arithmetic or heap-based commands, whereas PROVER-OVERAPPROX is overwhelmingly successful on programs with these features. This also shows the usefulness of overapproximation-based approach for proving non-termination.

Chapter 7

Conclusion

We have studied the problem of proving non-termination of programs and have presented three new methods for the same. We have also provided a thorough comparison of these methods along with previous methods.

In Chapter 3 we have introduced a new method of proving non-termination. The idea is to split the reasoning in two parts: a safety prover is used to prove that a loop in an underapproximation of the original program *never* terminates; meanwhile failed safety proofs are used to calculate the underapproximation. We have shown that non-determinism can be easily handled in our framework while previous tools often fail. Furthermore, we have shown that our approach leads to performance improvements against previous tools where they are applicable.

In Chapter 4 we have presented a novel Max-SMT-based technique for proving that programs do not terminate. The key notion of the approach is that of a *quasi-invariant*, which is a property such that if it holds at a location during execution once, then it continues to hold at that location from then onwards. The method considers one SCSG of the control flow graph at a time, and thanks to Max-SMT solving generates a quasi-invariant for each location. Weights of soft constraints guide the solver towards quasi-invariants that are also *edge-closing*, i.e., that forbid any transition exiting the SCSG. If an SCSG with edge-closing quasi-invariants is reachable, then the program is non-terminating. This last check is performed with an off-the-shelf reachability checker. We have reported experiments with encouraging results that show that a prototypical implementation of the proposed approach has often better efficacy than previous non-termination provers.

Overapproximation is the workhorse of program analysis. Unfortunately, over-

approximation can invalidate conventional techniques for proving non-termination. In Chapter 5 we have introduced the notion of a *live abstraction* to show how overapproximation can *help*, not hinder non-termination proving. The idea is to prove the existence of a *closed recurrence set* rather than simply a recurrence set. This modification in strategy allows us to use off-the-shelf overapproximating abstractions, leading to a new set of methods for proving non-termination of real programs. Our prototypical implementation of these ideas can prove non-termination of programs with non-linear arithmetic and heap-based commands whereas previous tools often fail.

The experimental evidence provided in Chapter 6 confirms the usefulness of non-termination proving methods presented in the thesis.

Bibliography

[ABAG11] Diego Esteban Alonso-Blas, Puri Arenas, and Samir Genaim. Handling non-linear operations in the value analysis of COSTA. In *Proc. BYTECODE '11*, 2011. One citation on page 71.

[ABEL12] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. Detecting fair non-termination in multithreaded programs. In *Proc. CAV '12*. Springer, 2012. One citation on page 18.

[BBH+12] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Thomas Vojnar. Programs with lists are counter automata. In *Proc. CAV '06*. Springer, 2012. One citation on page 73.

[BCDO06] Josh Berdine, Byron Cook, Dino Distefano, and Peter O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV '06*. Springer, 2006. 3 citations on pages 10, 86, and 87.

[BCF13] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*. Springer, 2013. 3 citations on pages 11, 62, and 82.

[BCHR13] Domagoj Babic, Byron Cook, Alan J. Hu, and Zvonimir Rakamaric. Proving termination of nonlinear command sequences. *Formal Asp. Comput.*, 25(3), 2013. One citation on page 11.

[BHMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009. One citation on page 47.

[BHRC07] Domagoj Babic, Alan J. Hu, Zvonimir Rakamaric, and Byron Cook. Proving termination by divergence. In *Proc. SEFM '07*. IEEE Computer Society, 2007. One citation on page 11.

[BK11] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Proc. CAV '11*. Springer, 2011. One citation on page 82.

[BLN+09] Cristina Borralleras, Salvador Lucas, Rafael Navarro-Marset, Enric Rodríguez-Carbonell, and Albert Rubio. Solving non-linear polynomial arithmetic via SAT modulo linear arithmetic. In *Proc. CADE '09*. Springer, 2009. 2 citations on pages 56 and 57.

[BMS05a] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *Proc. CAV '05*. Springer, 2005. One citation on page 11.

[BMS05b] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *Proc. ICALP '05*. Springer, 2005. One citation on page 11.

[BMS05c] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *Proc. CONCUR '05*. Springer, 2005. One citation on page 11.

[BMS05d] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In *Proc. VMCAI '05*. Springer, 2005. One citation on page 11.

[Bro15] James Brotherston. Personal communication. 2015. One citation on page 80.

[BSOG12] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. In *Proc. FoVeOOS '11*. Springer, 2012. 3 citations on pages 15, 17, and 83.

[CCF+14] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Proving nontermination via safety. In *Proc. TACAS '14*. Springer, 2014. One citation on page 19.

[CCG+05] Sagar Chaki, Edmund M. Clarke, Orna Grumberg, Joël Ouaknine, Natasha Sharygina, Tayssir Touili, and Helmut Veith. State/event software verification for branching-time specifications. In *Proc. IFM '05*. Springer, 2005. One citation on page 18.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011. 3 citations on pages 28, 32, and 42.

[CFNO14] Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Disproving termination with overapproximation. In *Proc. FMCAD '14*. IEEE, 2014. One citation on page 20.

[CGL+08] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *Proc. CAV '08*. Springer, 2008. One citation on page 11.

[CK13] Byron Cook and Eric Koskinen. Reasoning about nondeterminism in programs. In *Proc. PLDI '13*. ACM, 2013. 2 citations on pages 18 and 34.

[CKP14] Byron Cook, Heidy Khlaaf, and Nir Piterman. Faster temporal reasoning for infinite-state programs. In *Proc. FMCAD '14*. IEEE, 2014. One citation on page 18.

[CKRW13] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design*, 43(1):93–120, 2013. One citation on page 11.

[CKV11] Byron Cook, Eric Koskinen, and Moshe Y. Vardi. Temporal property verification as a program analysis task. In *Proc. CAV '11*. Springer, 2011. One citation on page 18.

[Cou05] Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Proc. VMCAI '05*. Springer, 2005. One citation on page 11.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*. ACM Press, 2006. One citation on page 11.

[CPR09] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3):369–387, 2009. One citation on page 11.

[CS02] Michael Colón and Henny Sipma. Practical methods for proving program termination. In *Proc. CAV '02*. Springer, 2002. One citation on page 11.

[CSS03] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV '03*. Springer, 2003. 3 citations on pages 44, 50, and 57.

[CSZ13] Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *Proc. TACAS '13*. Springer, 2013. 2 citations on pages 11 and 82.

[DGG00] Dennis Dams, Rob Gerth, and Orna Grumberg. A heuristic for the automatic generation of ranking functions. In *Workshop on Advances in Verification*, 2000. One citation on page 11.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. One citation on page 32.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*. Springer, 2008. One citation on page 83.

[EEG12] Fabian Emmes, Tim Enger, and Jürgen Giesl. Proving non-looping non-termination automatically. In *Proc. IJCAR '12*. Springer, 2012. One citation on page 18.

[FGKP85] Nissim Francez, Orna Grumberg, Shmuel Katz, and Amir Pnueli. Proving Termination of Prolog Programs. In *Proc. Logic of Programs '85*. Springer, 1985. One citation on page 11.

[GBE+14] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *Proc. IJCAR '14*. Springer, 2014. 2 citations on pages 17 and 83.

[GHM+08] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proc. POPL '08*. ACM, 2008. 11 citations on pages 16, 19, 26, 31, 58, 64, 67, 76, 77, 80, and 83.

[GST06] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*. Springer, 2006. One citation on page 11.

[GSV08] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proc. PLDI '08*. ACM Press, 2008. One citation on page 18.

[GTS05] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*. Springer, 2005. One citation on page 18.

[GWC06] Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A software model-checker for verification and refutation. In *Proc. CAV '06*. Springer, 2006. One citation on page 18.

[HLNR10] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. Alternation for termination. In *Proc. SAS '10*. Springer, 2010. One citation on page 62.

[JdM12] Dejan Jovanovic and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In *Proc. IJCAR '12*. Springer, 2012. 2 citations on pages 75 and 83.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. CAV '09*. Springer, 2009. 4 citations on pages 10, 37, 71, and 72.

[KSTW10] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *Proc. CAV '10*. Springer, 2010. One citation on page 11.

[LNO+14] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using max-smt. In *Proc. CAV '14*. Springer, 2014. 2 citations on pages 19 and 20.

[LORR13] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving Termination of Imperative Programs Using Max-SMT. In *Proc. FMCAD '13*. IEEE, 2013. 3 citations on pages 43, 61, and 86.

[LORR14] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Minimal-model-guided approaches to solving polynomial constraints and extensions. In *Proc. SAT '14*. Springer, 2014. One citation on page 57.

[LRR13] Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio. Smt-based array invariant generation. In *Proc. VMCAI '13*. Springer, 2013. One citation on page 43.

[Mag10] Stephen Magill. *Instrumentation Analysis: An Automated Method for Producing Numeric Abstractions of Heap-Manipulating Programs*. PhD thesis, 2010. One citation on page 73.

[McM06] Ken McMillan. Lazy abstraction with interpolants. In *Proc. CAV '06*. Springer, 2006. One citation on page 82.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proc. VSTTE '12*. Springer, 2012. One citation on page 83.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), 2006. 3 citations on pages 37, 71, and 80.

[MTLT10] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL '10*. ACM, 2010. 4 citations on pages 10, 72, 73, and 74.

[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. *TOPLAS*, 11(4), 1989. One citation on page 22.

[NO06] Robert Nieuwenhuis and Albert Oliveras. On SAT Modulo Theories and Optimization Problems. In *Proc. SAT '06*. Springer, 2006. 2 citations on pages 47 and 57.

[OBEG10] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, volume 6. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010. One citation on page 11.

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proc. CSL '01*. Springer, 2001. One citation on page 73.

[Pay08] Étienne Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theor. Comput. Sci.*, 403(2-3), 2008. One citation on page 18.

[PM09] Étienne Payet and Frédéric Mesnard. A non-termination criterion for binary constraint logic programs. *TPLP*, 9(2), 2009. One citation on page 17.

[PR04] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. VMCAI '04*. Springer, 2004. One citation on page 11.

[PR07] Andreas Podelski and Andrey Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *Proc. PADL '07*. Springer, 2007. One citation on page 11.

[PS09] Étienne Payet and Fausto Spoto. Experiments with non-termination analysis for Java Bytecode. *Electr. Notes Theor. Comput. Sci.*, 2009. 2 citations on pages 17 and 83.

[PTVF89] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge Univ. Press, 1989. One citation on page 83.

[SMP10] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010. One citation on page 83.

[TSWK11] Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *Proc. TACAS '11*. Springer, 2011. One citation on page 11.

[VR08] Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *Proc. TAP '08*. Springer, 2008. 2 citations on pages 17 and 83.