Precise Verification of C Programs

Matt Lewis

St Johns's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2014

Acknowledgements

I owe a great debt to all of the people who have supported me over the years, and especially to those who have helped me in the four years it took to produce this thesis. There is not nearly enough space here for me to thank everybody I need to, so I have to restrict myself to a few.

I couldn't have done this without the people who have spent so much time and patience to teach me how to do research. My supervisor, Daniel Kroening, who has been a constant source of inspiration and support. His non-stop optimism that we could get things to work, even when they seemed impossible, was crucial throughout my time here. Byron Cook, who first suggested I study in Oxford and who first taught me how to build a verification tool. Georg Weissenbacher, who taught me by his example how a good paper should be written.

My family: Rose, Chris and Tessa. You've always looked after me & I love you all.

Thanks to all of my friends who've made the last four years so much fun. In particular, I'd like to give shout outs to everyone from my college, the department and the miscellaneous other groups of awesome people I've run into here in Oxford. The names that jump to my mind right now are Adam Gammack, Adrastos Omissi, Alistair Donaldson, Amica Dall, Ant Hibbs, Ces Omissi, Chloe Omissi, Curt von Keyserlingk, David Landsberg, Duncan Palmer, Ed Greening, Emily Harrington, Gaelle Coullon, Ganesh Narayanaswamy, Ian Ashpole, Ian Cooper, Jacob Cook, Jade Alglave, Jake Trow, Jess Campbell, Justine Schluntz, La'akea Yoshida, Leo Haller, Leo Omissi, Madu Jayatunga, Malcolm Reynolds, Maria Friedmannova, Martin Brain, Nithum Thain, Ollie Harriman, Pascal Kesseli, Peter Schrammel, Richard Lau, Richard Stebbing, Rory Beard, Ross Haines, Sara Pereira, Sven Ernst, Tike Omissi, Tupac Shakur, Vincent Nimal and Struan Murray. Thanks guys, it's been great.

Finally, thanks to Cristina David who put up with me for so long and helped me so often with so many things. There were a lot of times when I would have given up, or just gone completely mad if it hadn't been for you.

I couldn't have done this without all of you.

Abstract

Most current approaches to software verification are one-sided – a safety prover will try to prove that a program is safe, while a bug-finding tool will try to find bugs. It is rare to find an analyser that is optimised for both tasks, which is problematic since it is hard to know in advance whether a program you wish to analyse is safe or not. The result of taking a one-sided approach to verification is false alarms: safety provers will often claim that safe programs have errors, while bug-finders will often be unable to find errors in unsafe programs.

Orthogonally, many software verifiers are designed for reasoning about idealised programming languages that may not have widespread use. A common assumption made by verification tools is that program variables can take arbitrary integer values, while programs in most common languages use fixed-width bitvectors for their variables. This can have a real impact on the verification, leading to incorrect claims by the verifier.

In this thesis we will show that it is possible to analyse C programs without generating false alarms, even if they contain unbounded loops, use non-linear arithmetic and have integer overflows. To do this, we will present two classes of analysis based on underapproximate loop acceleration and second-order satisfiability respectively.

Underapproximate loop acceleration addresses the problem of finding deep bugs. By finding closed forms for loops, we show that deep bugs can be detected without unwinding the program and that this can be done without introducing false positives or masking errors. We then show that programs accelerated in this way can be optimised by inlining trace automata to reduce their reachability diameter. This inlining allows acceleration to be used as a viable technique for proving safety, as well as finding bugs.

In the second part of the thesis, we focus on using second-order logic for program analysis. We begin by defining *second-order SAT*: an extension of propositional SAT that allows quantification over functions. We show that this problem is NEXPTIME-complete, and that it is polynomial time reducible to finite-state program synthesis. We then present a fully automatic, sound and complete algorithm for synthesising C programs from a specification written in C. Our approach uses a combination of bounded model checking, explicit-state model checking and genetic programming to achieve surprisingly good performance for a problem with such high complexity. We conclude by using second-order SAT to precisely and directly encode several program analysis problems including superoptimisation, de-obfuscation, safety and termination for programs using bitvector arithmetic and dynamically allocated lists.

Contents

Chapter 1

Introduction

Software verification is concerned with proving properties of computer programs. Two properties that we are often interested in proving are safety and liveness. The safety problem can be phrased informally as "are there any executions of this program in which something bad happens?" Dually, liveness asks the question "are there any executions in which nothing good happens?" Unfortunately, both of these properties are undecidable for general programs. In response to this problem of undecidability, software verification has restricted itself in certain dimensions, typically by limiting the class of programs to be analysed, or by allowing wrong answers to be reported. In this thesis, we will concern ourselves with the automatic, precise verification of C programs. For our purposes, an analysis is precise if it generates neither false alarms nor false claims of correctness.

Our work is initially motivated by the desire to analyse C programs for security problems. Some of the requirements for such analyses are that they must handle programs with very deep loops, they must correctly handle integer overflow and they must be able to find witnesses exhibiting errors whenever such an error exists. It is not enough for us to simply claim that a program contains an error, we must supply evidence. In addition to the previously stated goals of automation and precision, our secondary goals are to handle deep loops, integer overflows and to find concrete error witnesses.

The main difficulty in program analysis is the presence of loops hindering the computation of reachable states. This issue is addressed by computing either over- or under-approximations of the set of reachable states during the execution of a program. Typically, safety checkers are associated with over-approximating the set of reachable states by a safety invariant (i.e. the safety proof). The automatic inference of safety invariants is a challenging problem addressed by a multitude of research works among which those based on abstract interpretation [34, 35, 82] proved very successful.

```
y = 10;
x = 10;

while (x < 250000000) {
  x++;
  y++;
}

assert(x == y);
```

```
y = 10;
x = 11;

while (x < 250000000) {
  x++;
  y++;
}

assert(x == y);
```

Figure 1.1: Two programs with deep loops.

Such an analysis computes the least fixed point of a system of equations over a lattice. As the generated abstract fixed point is an over-approximation of the set of reachable states of the original program, the generated counterexample might be spurious, i.e. it can reach the error state according to an abstract semantics, but not in the concrete semantics.

On the other hand, approaches designed for bug finding, e.g. testing and bounded model checking [29], under-approximate the set of reachable states by examining a single trace or by performing loop unwinding, respectively. Consequently, they are not suitable for safety analysis since it is hard to guarantee that all of the relevant states have been examined. Both approaches' cost scales exponentially with the depth of the bug.

Approaches based on a combination of over- and under-approximations such as predicate abstraction and Lazy Abstraction with Interpolants (LAwI) try to bridge this gap and generate both safety proofs and concrete counterexamples. However, they are not optimised for finding errors as they can only detect counterexamples with deep loops after exploring numerous spurious and increasingly longer counterexamples.

To illustrate the shortcomings of the most common static analysis techniques, consider the programs in Figure 1.1. The program on the left is safe, while the program on the right is unsafe (the assertion will fail). The most common technique for proving program safety is abstract interpretation. A common arithmetic abstract domain is that of *intervals*, which for each variable tracks an interval containing the values that variable can take. For the program on the right, an abstract interpreter using intervals would conclude (after reaching a fixed point) that at the end of the loop, x is in the range $[11, 250000011]$ and that y is in the range $[10, 250000010]$. This

```
int x = nondet();
int y = nondet();

while (x < y) {
  x += 2;
}
```

```
unsigned x = nondet();
unsigned y = nondet();

while (x < y) {
  x += 2;
}
```

Figure 1.2: Programs whose behaviour depends on integer overflows.

is not enough information to conclude that $x = y$, and so the abstract interpreter will (correctly) fail to prove safety of this program. However, when analysing the left-hand program the intervals modelling x and y will be $[10, 250000010]$ and $[10, 250000010]$ respectively. Again, this is not enough information to conclude that $x = y$ and the abstract interpreter will *also* fail to prove safety for the program on the left. For these two programs, abstract intrepretation with intervals is unable to give us any useful information – it incorrectly tells us that the safe program has a bug, and is unable to provide us any evidence of the bug in the unsafe version. There are more complex abstract domains that can differentiate these two programs, in particular relational domains, but any abstract domain will suffer from imprecision of some kind that will lead to false alarms.

Static analysers using underapproximation do not fare much better on these examples. The most common underapproximate static analysers, bounded model checkers, will have to completely unwind the loops in order to analyse these programs. The resulting SAT instance will be very large, so large in fact that solving it will almost certainly take far too long to be practical, at least on a current (2014-era) computer. Running these programs through a bounded model checker on my 3 GHz machine with 8 GB of RAM rapidly exhausts the available memory trying to build the SAT formula. Hybrid approaches such as predicate abstraction and LAwI will be able to prove that the left-hand program is safe (by discovering the safety invariant $x = y$), but will very likely have to resort to unwinding in order to discover that the right-hand program is unsafe, which will cause the analysis to timeout.

Another source of imprecision when analysing C programs is arithmetic. Integer variables in C are implemented as fixed-width bitvectors, usually 32-bits or 64-bits wide. Arithmetic on these bitvectors can "overflow" – when the result of an operation is too large to store in a variable, it is truncated so that it does fit. For performance reasons, many analysers use mathematical integers to model program

```
list_t x, y, p, q;

p = x;
y = NULL;

while (p != NULL) {
  q = new();
  q->next = y;
  y = q;
  p = p->next;
}

p = x;

while (p != NULL) {
  p = p->next;
  y = y->next;
}

assert(y == NULL);
```

```
list_t x, y, p, q;

p = x;
y = new();

while (p != NULL) {
  q = new();
  q->next = y;
  y = q;
  p = p->next;
}

p = x;

while (p != NULL) {
  p = p->next;
  y = y->next;
}

assert(y == NULL);
```

Figure 1.3: Programs whose behaviour depends on properties of the heap.

variables. Mathematical integers do not have this wraparound behaviour and so the semantics of the program as understood by the analyser can differ from its semantics as implemented by the compiler and hardware, which can lead to imprecise analyses. Consider the programs in Figure 1.2: the program on the left will terminate for an value assigned nondeterministically to x and y. However, the program on the right will not terminate if y is assigned MAXINT and x is initially assigned an odd value, since the addition in the body of the loop will overflow before x can exceed y. An analysis based on mathematical integers will therefore not be able to differentiate these two programs in terms of their termination behaviour.

The situation becomes even more difficult when we add the heap into the mix. Programs using dynamically allocated data structures are extremely hard to analyse precisely, especially when their analysis requires reasoning about the shape of the heap in combination with arithmetic. The programs in Figure 1.3 illustrate this difficulty. Both of these programs construct a singly linked list in y and then iterate over it. In both programs, if the initial list x is circular, then the first loop will not terminate. If it does, the program on the left has constructed a list y that is the same length as the list x and so the second loop will proceed with no memory errors, terminate and then the assertion will pass. By contrast, the program on the right constructs a list y that is 1 cell longer than x, and so the final assertion will fail. In order to correctly recognise that the program on the left is safe while the one on the right is not, an analyser must be able to reason about the length of the lists, the shape of the lists, and the aliasing relationships amongst the pointers in the heap, which is difficult to do.

Each of these causes of imprecision (deep loops, integer overflows and the heap) is a headache by itself. In combination, they quickly lead to programs that are intractable for current verification techniques. In this thesis we will attempt to show that it is possible to analyse C progams with all of these features, without generating spurious warnings and without claiming that unsafe programs are safe.

1.1 Thesis Structure

This thesis is divided into two parts. In Part I, we will discuss a verification method based on *underapproximate loop acceleration*. The core idea of this technique is to identify looping paths for which we can compute precise symbolic closed forms. Combining this acceleration with BMC and interpolation based model checking, we will first show how to find deep bugs and then how to prove safety.

In Part II, we will present an extension of propositional SAT that we call *Second-Order SAT*. This generalisation allows us to quantify over *functions* as well as variables. After developing the theory of Second-Order SAT and proving some complexity bounds, we will show how to build a Second-Order SAT solver via a reduction to program synthesis. Having built this solver, we show how to use it for program analysis by encoding termination and safety of bitvector programs as Second-Order SAT. As part of our safety encoding we introduce *danger invariants*, which allow us to find deep bugs without unrolling loops. Finally, we present a theory of singly-linked lists with a decision procedure in NP. Since the decision procedure is in NP, it can be combined with Second-Order SAT to allow reasoning about programs using the heap.

The material in Part I is most naturally presented in terms of automata, while Part II is naturally presented in terms of logic. For ease of presentation, we begin each part with a brief section explaining the relevant background and defining notation.

1.2 Contributions

1.2.1 Acceleration

- We introduce underapproximate acceleration: a method for finding deep bugs in C programs. Acceleration does not introduce false positives, nor does it mask errors. It is bit-level accurate and can handle programs that make limited use of arrays. It can be freely combined with a variety of existing analyses including BMC and LAwI.

- We use trace automata as a means to optimise accelerated programs. Trace automata remove redundant traces from an accelerated program, which can significantly reduce the reachability diameter of the program. This diameter reduction can allow safety to be proved using just BMC, even for programs with unbounded loops.

- We implemented the combination of acceleration and trace automata as a source-to-source transformation. We experimentally evaluated the method using this implementation and found it to be better than state-of-the-art model checkers at analysing programs with deep loops. Our implementation has been released under an open source license as part of the CProver framework [66].

1.2.2 Second-order SAT

- We define the Second-Order SAT problem: an extension of propositional SAT that allows quantification over functions with propositional arguments.

- We prove that Second-Order SAT is NEXPTIME-complete.

- We provide a polynomial-time reduction from Second-Order SAT to finite-state program synthesis.

- We provide a novel algorithm for finite-state program synthesis, which uses a combination of symbolic model checking, explicit-state model checking and evolutionary search to efficiently and automatically synthesise programs.

- We prove that the runtime of our synthesis procedure is a function of the Kolmogorov complexity of the synthesis problem, and observe that many program analysis tasks can be reduced to Second-Order SAT problems that often have low Kolmogorov complexity.

- We show how many program analysis tasks can be directly encoded as Second-Order SAT problems. We provide a detailed exposition of how termination and non-termination of bitvector programs can be encoded as Second-Order SAT. The resulting Second-Order SAT encoding is precise – if a proof of termination or non-termination is found, it is correct; we never incorrectly claim that a program terminates or fails to.

- We prove that our proof language is expressive enough to prove termination or non-termination for *every* bitvector program, showing that our termination analysis is both sound and complete.

- We introduce a formulation that enables us to prove the presence of arbitrarily deep bugs without unrolling loops: *danger invariants*. This technique is distinct from our previously introduced notion of underapproximate acceleration.

- We propose the use of *second-order tautologies* for software analysis. These are second-order formulae that are theorems by construction, which we show to be a useful method for reducing the Kolmogorov complexity of proofs of program properties.

- We implemented our solver for Second-Order SAT and evaluated it on a selection of problems generated from a wide range of domains including superoptimisation, QBF solving and termination analysis. We show that the termination analysis built this way is more precise than state-of-the-art termination provers and that its performance is practical.

1.2.3 Heap

- We develop a theory of singly linked lists that enables us to analyse programs manipulating dynamically allocated singly linked lists. Our theory is able to soundly and precisely reason about lists with cycles, as well as arbitrary, unspecified sharing.

- We prove that our theory has a small model property, which we use to build a decision procedure based on a reduction to SAT.

- The reduction to SAT enables our theory to be combined with Second-Order SAT in order to automatically infer invariants and ranking functions for heap manipulating programs.

- We implemented our decision procedure and show that the theory is expressive enough to encode total correctness proofs for non-trivial programs, requiring reasoning about a combination of shape properties and non-linear arithmetic over bitvectors.

1.3 Research Hypothesis

The received wisdom in software verification tells us that we must make compromises when analysing programs. In order to get watertight safety proofs, we must accept a certain rate of false alarms; in order to find real bugs we must sacrifice completeness and the efficiency of symbolic methods; in order to use powerful logical and symbolic methods, we must ignore tricky, real world behaviours such as integer overflows.

In this thesis we will examine some of these compromises and try to reduce the number of limitations we need to accept. Our hypothesis is that it is possible to analyse C progams with loops, integer overflows and dynamically allocated data structures, while at the same time generating no spurious warnings or claiming that unsafe programs are safe.

Part I

Underapproximate Acceleration

Chapter 2

Overview and Preliminaries

Collaborators *The bulk of the material in this part comes from [65, 67] which were written in conjunction with Georg Weissenbacher.*

In this part, we will make use of underapproximate acceleration to analyse programs. The key idea is to preprocess the program under analysis by finding *closed forms* for loops which capture the effect of executing the loop n times. In other words we find a symbolic expression for the transitive closure of the loop. Since a loop may have several paths through its body, we will be accelerating one path at a time and combining the resulting accelerators into one program. The underapproximation stems from this idea of considering a single path through the loop at a time – one path underapproximates the single step transition relation of the loop, and so by accelerating that path we arrive at an underapproximation for the transitive closure of the loop as a whole. Once we have an accelerator in hand, we union it back in to the original program by adding an extra looping path. The net effect of this transformation is that we create a program that has exactly the same set of reachable states as the first program, but in which a single unwinding of the instrumented loop captures the behaviour of an *arbitrary* number of unwindings of the original loop. In keeping with the theme of this thesis, the analyses built on underapproximate acceleration are precise – they do not generate false alarms and they are sound when they report safety.

In contrast to related work [15,44,58], our technique is bit-level accurate, supports assignments to arrays and arbitrary conditional branching by computing quantified conditionals. As the computational cost of analysing programs with quantifiers is high, we introduce two novel techniques for summarising certain conditionals without quantifiers. The key insight is that many conditionals in programs (e.g., loop exit conditions such as $\texttt{i} \leq 100$ or even $\texttt{i} \neq 100$) exhibit a certain monotonicity property that allows us to drop quantifiers.

In Chapter 3 we show how generate accelerators, instrument them into the program under analysis and use them to uncover deep bugs. Our approximation can be combined soundly with a broad range of verification engines, including predicate abstraction, lazy abstraction with interpolation [81], and BMC [29]. To demonstrate this versatility, we combined our technique with lazy abstraction and the CBMC [29] model checker. We evaluated the resulting tool on a large suite of benchmarks known to contain deep paths, demonstrating our ability to efficiently detect deep counterexamples in C programs that manipulate arrays.

In Chapter 4 we further exploit underapproximate acceleration to prove safety without using abstraction or invariant generation. We do this by making use of the fact that BMC is able to prove safety once the unwinding bound exceeds the reachability diameter of the model [13, 69]. The diameter of non-trivial programs is however unmanagably large in most cases. Furthermore, even when the diameter is small, it is often computationally expensive to determine, as the problem of computing the exact diameter is equivalent to a 2-QBF instance. We use underapproximate accelerators in conjunction with *trace automata* to reduce the diameter of the program under analysis without eliminating any reachable states.

2.1 Background and Notation

Let Stmts be the set of statements of a simple programming language as defined in Table 2.1a, where Exprs and $\mathbb{B}\text{-}\mathsf{Exprs}$ denote expressions and predicates over the program variables Vars, respectively. Assumptions are abbreviated by $[B]$, and assertions are modeled using assumptions and error locations. For brevity, we omit array accesses. We assume that different occurrences of statements are distinguishable (using the program locations). The semantics is provided by the weakest liberal precondition *wlp* as defined in [83]. Programs are represented using control-flow automata. In accordance with [83], $\pi_1 \Box \pi_2$ represents the non-deterministic choice between two paths, i.e., $$\begin{tikzcd} \circ \arrow[r, bend left, "\pi_1"] \arrow[r, bend right, "\pi_2"'] & \circ \end{tikzcd}$$. The commutative operator $\Box$ is extended to sets of paths in the usual manner.

Definition 1 (CFA)**.** A *control-flow automaton* is a directed graph $\langle V, E, v_0 \rangle$, where V is a finite set of vertices, $E \subseteq (V \times \mathsf{Stmts} \times V)$ is a set of edges, and $v_0 \in V$ is the initial vertex. We write $v \xrightarrow{\mathsf{stmt}} u$ if $\langle u, \mathsf{stmt}, v \rangle \in E$.

Table 2.1: Program Statements and Traces

$$\texttt{stmt} ::= \texttt{x} := e \mid [B] \mid \mathsf{skip}, \quad \texttt{x} \in \mathsf{Vars},\ e \in \mathsf{Exprs},\ B \in \mathbb{B}\text{-}\mathsf{Exprs}$$

$$\begin{array}{lcl} wlp(\texttt{x} := e, P) & \overset{\text{def}}{=} & P[e/\texttt{x}] \\ wlp([B], P) & \overset{\text{def}}{=} & B \Rightarrow P \\ wlp(\mathsf{skip}, P) & \overset{\text{def}}{=} & P \end{array}$$

(a) Syntax and Semantics of Statements

$$\begin{array}{rcl} [\![\mathsf{stmt}]\!] & \overset{\text{def}}{=} & \neg wlp(\mathsf{stmt}, \bigvee_{\texttt{x}\in\mathsf{Vars}} \texttt{x} \neq \texttt{x}') \\ \mathsf{id} & \overset{\text{def}}{=} & [\![\mathsf{skip}]\!] \\ [\![\mathsf{stmt}_1;\mathsf{stmt}_2]\!] & \overset{\text{def}}{=} & [\![\mathsf{stmt}_1]\!] \circ [\![\mathsf{stmt}_2]\!] \\ [\![\mathsf{stmt}^n]\!] & \overset{\text{def}}{=} & [\![\mathsf{stmt}]\!]^n, \end{array}$$

$$\text{where} \quad \begin{array}{rcl} \mathsf{stmt}^n & \overset{\text{def}}{=} & \mathsf{stmt};(\mathsf{stmt}^{(n-1)}), \\ [\![\mathsf{stmt}]\!]^n & \overset{\text{def}}{=} & [\![\mathsf{stmt}]\!] \circ ([\![\mathsf{stmt}]\!]^{(n-1)}), \end{array} \quad \begin{array}{rcl} \mathsf{stmt}^0 & \overset{\text{def}}{=} & \varepsilon \\ [\![\mathsf{stmt}]\!]^0 & \overset{\text{def}}{=} & \mathsf{id} \end{array}$$

(b) Transition Relations for Statements and Traces

A program state σ is a total function assigning a value to each program variable in Vars. States denotes the set of program states. A transition relation $T \subseteq \mathsf{States} \times \mathsf{States}$ associates states with their successor states. Given Vars, let Vars' be a corresponding set of primed variables encoding successor states. The symbolic transition relation for a statement or trace is a predicate over $\mathsf{Vars} \cup \mathsf{Vars}'$ and can be derived using *wlp* as indicated in Table 2.1b (cf. [40]). We write $\langle \sigma, \sigma' \rangle \in [\![\mathsf{stmt}]\!]$ if $[\![\mathsf{stmt}]\!]$ evaluates to true under σ and σ' (i.e., $\sigma, \sigma' \models [\![\mathsf{stmt}]\!]$). A trace π is *feasible* if there exist states σ, σ' such that $\langle \sigma, \sigma' \rangle \in [\![\pi]\!]$.

Given a CFA $P \overset{\text{def}}{=} \langle V, E, v_0 \rangle$, a trace $\pi \overset{\text{def}}{=} \mathsf{stmt}_i; \mathsf{stmt}_{i+1}; \cdots \mathsf{stmt}_n$ (where $v_{j-1} \xrightarrow{\mathsf{stmt}_j} v_j$ for $i < j \leq n$) of length $|\pi| = n - i + 1$ is *looping* (with head v_i) iff $v_i = v_n$, and *accepted* by the CFA iff it is feasible and $v_i = v_0$ (The language $\mathcal{L}_P$ over the alphabet Stmts accepted by the CFA P is prefix-closed.)

A state σ is *reachable* from an initial state σ_0 iff there exists a trace π accepted by the CFA such that $\langle \sigma_0, \sigma \rangle \in [\![\pi]\!]$. The reachability diameter [13, 69] of a transition relation is the smallest number of steps required to reach all reachable states:

Definition 2 (Reachability Diameter)**.** Given a CFA with initial state σ_0, the *reachability diameter* is the smallest n such that for every state σ reachable from σ_0 there exists a trace π of length at most n accepted by the CFA with $\langle \sigma_0, \sigma \rangle \in [\![\pi]\!]$.

To show that a CFA does not violate a given safety (or reachability) property, it is sufficient to explore all traces whose length does not exceed the reachability diameter. In the presence of looping traces, however, the reachability diameter of a program can be infinitely large.

Acceleration [15, 18, 44] is a technique to compute the reflexive transitive closure $[\![\pi]\!]^* \overset{\text{def}}{=} \bigcup_{i=0}^{\infty} [\![\pi]\!]^i$ for a looping path π. Equivalently, $[\![\pi]\!]^*$ can be expressed as $\exists i \in \mathbb{N}_0 \,.\, [\![\pi]\!]^i$. The aim of acceleration is to express $[\![\pi]\!]^*$ in a decidable fragment of logic. In general, this is not possible, even if $[\![\pi]\!]$ is defined in a decidable fragment of integer arithmetic such as Presburger arithmetic.

Definition 3 (Accelerated Transitions)**.** Given a looping trace π, we say that $\hat{\pi}$ is an *accelerator* for π if

$$[\![\widehat{\pi}]\!] \equiv \exists i \in \mathbb{N}_0 \,.\, [\![\pi]\!]^i \,.$$

Chapter 3

Finding Bugs with Under-Approximate Loop Acceleration

3.1 Introduction

The generation of *diagnostic counterexamples* is a key feature of model checking. Counterexamples serve as witness for the refutation of a property, and are an invaluable aid to the engineer for understanding and repairing the fault.

Counterexamples are particularly important in software model checking, as bugs in software frequently require thousands of transitions to be executed, and are thus difficult to reproduce without the help of an explicit error trace. Existing software model checkers, however, fail to scale when analysing programs with bugs that involve many iterations of a loop. The primary reason for the inability of many existing tools to discover such "deep" bugs is that exploration is performed in a breadth-first fashion: the detection of an unsafe execution traversing a loop involves the repeated refutation of increasingly longer spurious counterexamples. The analyser first considers a potential error trace with one loop iteration, only to discover that this trace is infeasible. As consequence, the analyser will increase the search depth, usually by considering one further loop iteration. In practice, the computational effort required to discover an assertion violation thus grows exponentially with the depth of the bug.

Notably, the problem is not limited to procedures based on abstraction, such as predicate abstraction or abstraction with interpolants. Bounded Model Checking (BMC) is optimised for discovering bugs up to a given depth k, but the computational cost grows exponentially in k.

The contribution of this chapter is a new technique that enables scalable detection of deep bugs. We transform the program by adding a new, auxiliary path to loops that summarises the effect of a parametric number of iterations of the loop. Similar to acceleration, which captures the exact effect of arbitrarily many iterations of an integer relation by computing its reflexive transitive closure in one step [15, 44, 58], we construct a summary of the behaviour of the loop. By symbolically bounding the number of iterations, we obtain an *under-approximation* which is sound with respect to the bit-vector semantics of programs. Thus, we avoid false alarms that might be triggered by modeling variables as integers.

In contrast to related work, our technique supports assignments to arrays and arbitrary conditional branching by computing quantified conditionals. As the computational cost of analysing programs with quantifiers is high, we introduce two novel techniques for summarising certain conditionals without quantifiers. The key insight is that many conditionals in programs (e.g., loop exit conditions such as $\texttt{i} \leq 100$ or even $\texttt{i} \neq 100$) exhibit a monotonicity property that allows us to drop quantifiers.

Our approximation can be combined soundly with a broad range of verification engines, including predicate abstraction, lazy abstraction with interpolation [81], and bounded software model checking [29]. To demonstrate this versatility, we combined our technique with lazy abstraction and the CBMC [29] model checker. We evaluated the resulting tool on a large suite of benchmarks known to contain deep paths, demonstrating our ability to efficiently detect deep counterexamples in C programs that manipulate arrays.

3.2 Outline

3.2.1 A Motivating Example

A common characteristic of many contemporary symbolic software model checking techniques (such as counterexample-guided abstraction refinement with predicate abstraction [5, 56], lazy abstraction with interpolants [81], and bounded model checking [29]) is that the computational effort required to discover an assertion violation may increase exponentially with the length of the corresponding counterexample path (c.f. [70]). In particular, the detection of assertion violations that require a large number of loop iterations results in the enumeration of increasingly longer *spurious* counterexamples traversing that loop. This problem is illustrated by the following example.

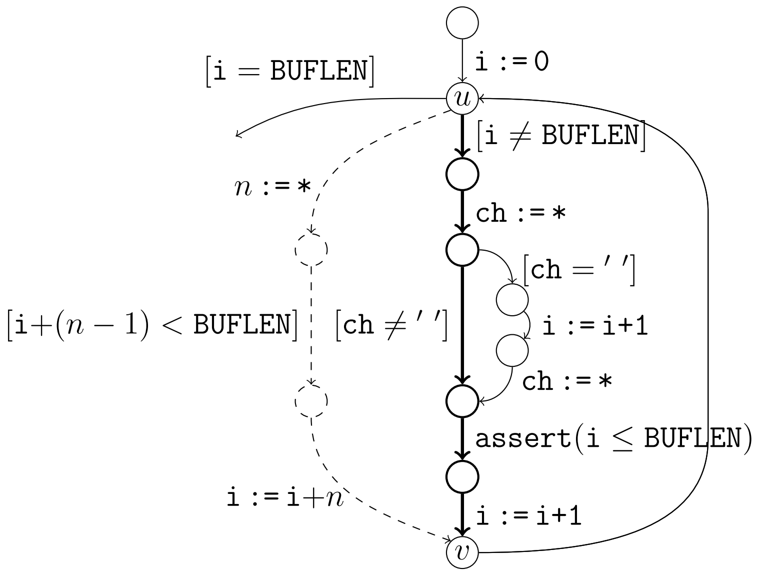


Figure 3.1: CFG with path π (bold) and approximated path $\widehat{\pi}$ (dashed)

Example 1. Figure 3.1 shows a program fragment derived from code permitting a buffer overflow (detected by the assertion) to occur in the n^{th} iteration of the loop if `i` reaches $(\texttt{BUFLEN} - 1)$ and the branch $[\texttt{ch} = \texttt{' '}]$ is taken in the $(n-1)^{\text{th}}$ iteration. The verification techniques mentioned above explore the paths in order of increasing length. The shortest path that reaches the assertion does not violate it, as

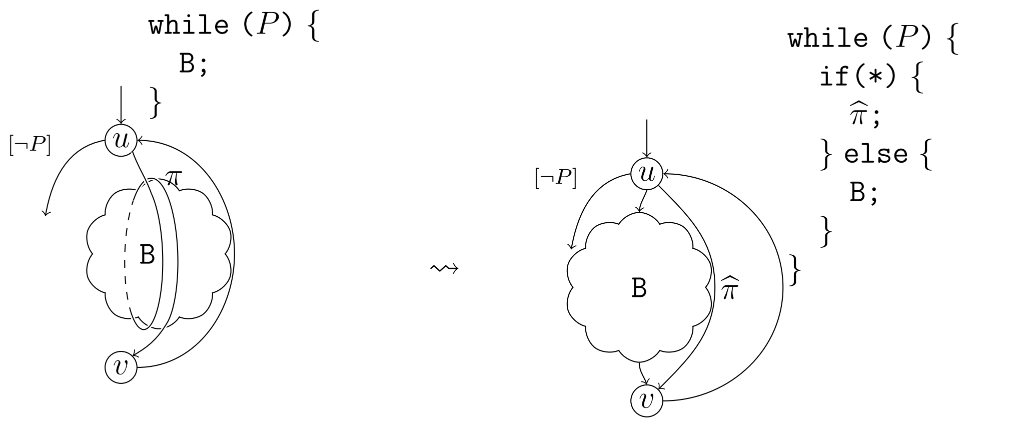
$$sp((\texttt{i} := 0; [\texttt{i} \neq \texttt{BUFLEN}\}]; \texttt{ch} := *; [\texttt{ch} \neq \texttt{' '}]), \mathsf{T}) \quad \Rightarrow \quad (\texttt{i} \leq \texttt{BUFLEN})\,.$$

In a predicate abstraction or lazy abstraction framework, this path represents the first in a series of spurious counterexamples of increasing length. Let π denote the path emphasised in Figure 3.1, which traverses the loop once. The verification tool will generate a family of spurious counterexamples with the prefixes `i := 0;` π^n (where $0 < n \leq \frac{\texttt{BUFLEN}}{2}$) before it detects a path long enough to violate the assertion. Each of these paths triggers a computationally expensive refinement cycle. Similarly, a bounded model checker will fail to detect a counterexample unless the loop bound is increased to $\frac{\texttt{BUFLEN}}{2} + 1$.

The iterative exploration of increasingly deeper loops primarily delays the detection of assertion violations (c.f. [70]), but can also result in a diverging series of interpolants and predicates if the program is safe (see [61]).

3.2.2 Approximating Paths with Loops

We propose a technique that aims at avoiding the enumeration of paths with an insufficient number of loop iterations. Our approach is based on the insight that the


```
while (P) {
  B;
}
```

```
while (P) {
  if(*) {
    π̂;
  } else {
    B;
  }
}
```

Detect path π that repeatedly traverses the loop body B

Augment loop body with a branch containing the *approximation* $\widehat{\pi}$

Figure 3.2: Approximating the natural loop with head u and back-edge $v \to u$. Path π is a path traversing the body B at least once, and may take different branches in B in subsequent iterations.

refutation of spurious counterexamples containing a sub-path of the form π^n is futile if there exists an n large enough to permit an assertion violation. We add an auxiliary path that bypasses the original loop body and represents the effect of π^n for a range of n (detailed later in the chapter). Our approach comprises the following steps:

1. We sensitise an existing tool to detect paths π that repeatedly traverse the loop body B (as illustrated in the left half of Figure 3.2). We emphasise that π may span more than one iteration of the loop, and that the branches of B taken by π in different iterations may vary.

2. We construct a path $\widehat{\pi}$ whose behaviour *under-approximates* $\Box\{\pi^n \mid n \geq 0\}$. This construction does not correspond to acceleration in a strict sense, since $\widehat{\pi}$ (as an under-approximation) does not necessarily represent an arbitrary number of loop iterations. Section 3.3 describes techniques to derive $\widehat{\pi}$.

3. By construction, the assumptions in $\widehat{\pi}$ may contain universal quantifiers ranging over an auxiliary variable which encodes the number of loop iterations. In Section 3.4, we discuss two cases in which (some of) these quantifiers can be eliminated, namely (a) if the characteristic function of the predicate $\neg wlp(\pi^n, \mathsf{F})$ is *monotonic* in the number of loop iterations n, or (b) if π^n modifies an array and the indices of the modified array elements can be characterised by means

of a quantifier-free predicate. We show that in certain cases condition (a) can be met by splitting π into several separate paths.

4. We augment the control flow graph with an additional branch of the loop containing $\widehat{\pi}$ (Figure 3.2, right). Section 3.5 demonstrates empirically how this program transformation can accelerate the detection of bugs that require a large number of loop iterations.

The following example demonstrates how our technique accelerates the detection of the buffer overflow of Example 1.

Example 2. Assume that the verification tool encounters the node u in Figure 3.1 a second time during the exploration of a path (u is the head of a natural loop with back-edge $v \to u$). We conclude that there exists a family of (sub-)paths π^n induced by the number n of loop iterations. The repeated application of the strongest postcondition to the parametrised path π^n for an increasing n gives rise to a recurrence equation $\mathtt{i}^{\langle n\rangle} = \mathtt{i}^{\langle n-1\rangle} + 1$ (for clarity, we work on a sliced path omitting statements referring to `ch`):

$$\begin{array}{rcl}
sp(\pi^1, \mathsf{T}) & = & \exists \mathtt{i}^{\langle 0\rangle} \,.\, (\mathtt{i}^{\langle 0\rangle} < \mathtt{BUFLEN}) \wedge (\mathtt{i} = \mathtt{i}^{\langle 0\rangle} + 1) \\
sp(\pi^2, \mathsf{T}) & = & \exists \mathtt{i}^{\langle 0\rangle}, \mathtt{i}^{\langle 1\rangle} \,.\, (\mathtt{i}^{\langle 0\rangle} < \mathtt{BUFLEN}) \wedge (\mathtt{i}^{\langle 1\rangle} < \mathtt{BUFLEN}) \wedge \\
 & & \qquad (\mathtt{i}^{\langle 1\rangle} = \mathtt{i}^{\langle 0\rangle} + 1) \wedge (\mathtt{i} = \mathtt{i}^{\langle 1\rangle} + 1) \\
 & \vdots & \\
sp(\pi^n, \mathsf{T}) & = & \exists \mathtt{i}^{\langle 0\rangle} \ldots \mathtt{i}^{\langle n-1\rangle} \,.\, \left(\bigwedge_{j=0}^{n-1} (\mathtt{i}^{\langle j\rangle} < \mathtt{BUFLEN}) \wedge (\mathtt{i}^{\langle j+1\rangle} = \mathtt{i}^{\langle j\rangle} + 1) \right)
\end{array}$$

where $\mathtt{i}^{\langle n\rangle}$ in the last line represents `i` after the execution of π^n. This recurrence equation can be put into its equivalent closed form $\mathtt{i}^{\langle n\rangle} = \mathtt{i}^{\langle 0\rangle} + n$. By assigning n a (positive) non-deterministic value, we obtain the approximation (which happens to be exact in this case):

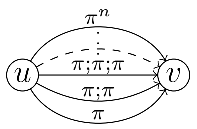
$$\widehat{\pi} \quad = \quad n := *; [\forall j \in [0, n) \,.\, \mathtt{i} + j < \mathtt{BUFLEN}]; \mathtt{i} := \mathtt{i} + n \quad .$$

Let us ignore arithmetic over- or under-flow for the time being (this topic is addressed in Section 3.3.4). We can then observe the following: if the predicate $\mathtt{i} + j < \mathtt{BUFLEN}$ is true for $j = n - 1$, then it must be true for any $j < n - 1$, i.e., the characteristic function of the predicate is *monotonic* in its parameter j. It is therefore possible to eliminate the universal quantifier and replace the assumption in $\widehat{\pi}$ with $(\mathtt{i} + (n-1) < \mathtt{BUFLEN})$. The dashed path in Figure 3.1 illustrates the corresponding modification of the original program. The resulting transformed program permits the violation of the assertion in the original loop body after a single iteration of $\widehat{\pi}$ (corresponding to `BUFLEN-1` iterations of π).

The following presents techniques to compute the under-approximation $\widehat{\pi}$.

3.3 Under-Approximation Techniques

This section covers techniques to compute under-approximations $\widehat{\pi}$ of $\bigsqcup\{\pi^n \,|\, n \geq 0\}$ such that $\widehat{\pi}$ is a condensation of the CFG fragment to the right. Formally, we only require that $sp(\widehat{\pi}, P) \Rightarrow \exists n \in \mathbb{N} \,.\, sp(\pi^n, P)$ for all P.



The construction of $\widehat{\pi}$ has two aspects. Firstly, we need to make sure that all variables modified in $\widehat{\pi}$ are assigned values consistent with π^n for a non-deterministic choice of n. Secondly, $\widehat{\pi}$ must only allow choices of n for which $\neg wlp(\pi^n, \mathsf{F})$ is satisfiable, i.e., the corresponding path π^n must be *feasible*.

Our approximation technique is based on the observation that the sequence of assignments in π^n to a variable $\mathtt{x} \in \mathtt{X}$ corresponds to a recurrence equation (c.f. Example 2). The goal is to derive an equivalent *closed* form $\mathtt{x} := f_{\mathtt{x}}(\mathtt{X}, n)$. While there is a range of techniques to solve recurrence equations, we argue that it is sufficient to consider closed-form solutions that have the form of low-degree polynomials. The underlying argument is that a super-polynomial growth of variable values typically leads to an arithmetic overflow after a small number of iterations, which can be detected at low depth using conventional techniques.

The following sub-section focuses on deriving closed forms from a sequence of assignments to scalar integer variables, leaving conditionals aside. Section 3.3.2 covers assignments to arrays. Conditionals and path feasibility are addressed in Section 3.3.3. Section 3.3.4 addresses bit-vector semantics and arithmetic overflow.

3.3.1 Computing Closed Forms of Assignments

Syntactic Matching.

A simple technique to derive closed forms is to check whether the given recurrence equation matches a pre-determined format. In [70], Weissenbacher applies the following scheme:

$$\mathtt{x}^{\langle 0\rangle} = \alpha, \quad \mathtt{x}^{\langle n\rangle} = \mathtt{x}^{\langle n-1\rangle} + \beta + \gamma \cdot n \quad \rightsquigarrow \quad \mathtt{x}^{\langle n\rangle} = \alpha + \beta n + \gamma \frac{n \cdot (n+1)}{2}\,, \tag{3.1}$$

where $n > 0$ and α, β, and γ are numeric constants or loop-invariant *symbolic* expressions and $\mathtt{x}$ is the variant. This technique is computationally cheap and sufficient to construct the closed form $\mathtt{i}^{\langle n\rangle} = \mathtt{i}^{\langle 0\rangle} + n$ of the recurrence equation $\mathtt{i}^{\langle n\rangle} = \mathtt{i}^{\langle n-1\rangle} + 1$ derived from the assignment $\mathtt{i} := \mathtt{i} + 1$ in Example 2.

Constraint-based Acceleration.

The disadvantage of a syntax-based approach is that it is limited to assignments following a simple pattern. Moreover, the technique is contingent on the syntax of the program fragment and may therefore fail even if there *exists* an appropriate polynomial representing the given assignments. In this section, we present an alternative technique that relies on a constraint solver to identify the coefficients of the polynomial $f_{\mathtt{x}}$.

Let $\mathtt{X}$ be the set $\{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ of variables in π. (In the following, we omit the braces $\{\}$ if clear from the context.) As previously, we start with the assumption that for each variable $\mathtt{x}$ modified in π, there is a low-degree polynomial in n

$$f_{\mathtt{x}}(\mathtt{X}^{\langle 0\rangle}, n) \overset{\text{def}}{=} \sum_{i=1}^{k} \alpha_i \cdot \mathtt{x}_i^{\langle 0\rangle} + \left(\sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathtt{x}_i^{\langle 0\rangle} + \alpha_{(2\cdot k+1)}\right) \cdot n + \alpha_{(2\cdot k+2)} \cdot n^2 \tag{3.2}$$

over the initial variables $\mathtt{x}_1^{\langle 0\rangle}, \ldots, \mathtt{x}_k^{\langle 0\rangle}$ which accurately represents the value assigned to $\mathtt{x}$ in π^n (for $n \geq 1$). In other words, for each variable $\mathtt{x} \in \mathtt{X}$ modified in π, we assume that the following Hoare triple is valid:

$$\{\bigwedge_{i=1}^{k} \grave{}\mathtt{x}_i = \mathtt{x}_i\}\ \pi^n\ \{\mathtt{x} = f_{\mathtt{x}}(\grave{}\mathtt{x}_1, \ldots, \grave{}\mathtt{x}_k, n)\} \tag{3.3}$$

For each $\mathtt{x} \in \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ we can generate $2 \cdot k + 2$ distinct assignments to $\mathtt{x}_1^{\langle 0\rangle}, \ldots, \mathtt{x}_k^{\langle 0\rangle}$, and n in (3.2) which determine a system of linearly independent equations over α_i, $0 < i \leq 2 \cdot k + 2$. If a solution to this system of equations exists, it *uniquely* determines the parameters $\alpha_1, \ldots, \alpha_{2\cdot k+2}$ of the polynomial $f_{\mathtt{x}}$ for $\mathtt{x}$. We will now examine the details of this construction and prove that it allows us to generate polynomial closed forms.

Lemma 4. *A set of vectors $\mathcal{X} = \{\vec{x}_1, \ldots, \vec{x}_n\}$ in vector space $\mathcal{V}$ is linearly independent if a projection of $\mathcal{X}$ onto a subspace $\mathcal{W}$ is linearly independent.*

Proof. Let $\mathcal{Y} = \{\vec{y}_1, \ldots, \vec{y}_m\}$ be the projection of $\mathcal{X}$ onto $\mathcal{W}$. Assume for contradiction that $\mathcal{X}$ is linearly dependent, then there is a set of scalars $a_1, \ldots, a_n$ such that $\sum_i a_i\vec{x}_i = \mathbf{0}$. But when we project onto $\mathcal{W}$ we have $\sum_i a_i\vec{y}_i = \mathbf{0}$, contradicting the assumption that $\mathcal{Y}$ is linearly independent. □

Theorem 5. *We can uniquely determine the coefficients for a polynomial over k variables by evaluating the loop at $2k + 2$ points with $n \leq 2$.*

Proof. Our polynomial is of the form

$$\sum_{i=1}^{k} \alpha_i \cdot \mathtt{x}_i + \sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathtt{x}_i \cdot n + \alpha_{(2\cdot k+1)} \cdot n + \alpha_{(2\cdot k+2)} \cdot n^2$$

There are $2 \cdot k + 2$ undetermined coefficients α_i $(1 \leq i \leq 2k+2)$ that we need to find. We need to generate a system of $2k+2$ linearly independent equations to uniquely fix these coefficients. This is equivalent to finding a set of $2k+2$ vectors

$$(x_{(i,1)}, \ldots, x_{(i,k)}, n_i) \quad \text{where } 1 \leq i \leq 2 \cdot k + 2$$

such that the set

$$\{(x_{(i,1)}, \ldots, x_{(i,k)}, x_{(i,1)} \cdot n, \ldots, x_{(i,k)} \cdot n, n_i, n_i^2) \,|\, 1 \leq i \leq 2 \cdot k + 1\}$$

is linearly independent. We generate this set inductively.

Basis: For $k = 1$, the set generated by

$$\begin{array}{ccc} (x_{(1,1)} = 1, n_1 = 0) & & (x_{(1,1)} = 1, x_{(1,1)} \cdot n = 0, n_1 = 0, n_1^2 = 0) \\ (0,1) & \text{is} & (0,0,1,1) \\ (1,1) & & (1,1,1,1) \\ (1,2) & & (1,2,2,4) \end{array}$$

which is linearly independent.

Induction: Assume we have a linearly independent set of $2 \cdot k + 2$ equations for k variables. We can extend the vectors by setting $x_{k+1} = 0$ in the vectors with $n = 0$ and $n = 1$, and by setting $x_{k+1} = 1$ in the vector with $n = 2$. By Lemma 4 this maintains linear independence.

Subsequently, we add two new vectors generated by

$$\begin{array}{l} (n_{(2\cdot k+3)} = 0, x_{(2\cdot k+3,k+1)} = 1, x_{2\cdot k+3, j\neq k+1} = 0) \quad \text{and} \\ (n_{(2\cdot k+4)} = 1, x_{(2\cdot k+4,k+1)} = 1, x_{(2\cdot k+4, j\neq k+1)} = 0)\,. \end{array}$$

The resulting $2 \cdot k + 4$ vectors are still linearly independent, which can be seen by projecting onto the space $(x_{k+1}, x_{k+1} \cdot n, n, n^2)$ – the only way to generate the $\mathbf{0}$ vector is by taking combinations of vectors all of which have $x_{k+1} = 0$. But that set is a subset of the $2 \cdot k + 2$ equations we started with, which are linearly independent, and so the extended set is also linearly independent by Lemma 1. So we have $2 \cdot k + 4 = 2 \cdot (k+1) + 2$ linearly independent vectors and the induction is complete. $\square$

In particular, the satisfiability of the encoding from which we derive the assignments guarantees that (3.3) holds for $0 \leq n \leq 2$. For larger values of n, we check the validity of (3.3) with respect to each $f_{\mathtt{x}}$ by means of induction. The validity of (3.3) follows (by induction over the length of the path π^n) from the validity of the base case established above, the formula (3.4) given below (which can be easily checked using a model checker or a constraint solver), and Hoare's rule of composition:

$$\{\bigwedge_{i=1}^{k} ({}^{\backprime}\mathtt{x}_i = \mathtt{x}_i) \wedge \mathtt{x} = f_{\mathtt{x}}({}^{\backprime}\mathtt{x}_1, \ldots, {}^{\backprime}\mathtt{x}_k, n)\}\ \pi\ \{\mathtt{x} = f_{\mathtt{x}}({}^{\backprime}\mathtt{x}_1, \ldots, {}^{\backprime}\mathtt{x}_k, n+1)\} \tag{3.4}$$

If for one or more $\mathtt{x} \in \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ our technique fails to find valid parameters $\alpha_1, \ldots, \alpha_{2\cdot k+2}$, or the validity check for $f_{\mathtt{x}}$ fails, we do not construct $\widehat{\pi}$.

Remark. The construction of under-approximations is not limited to the two techniques discussed above and can be based on other (and potentially more powerful) recurrence solvers.

3.3.2 Assignments to Arrays

Buffer overflows constitute a prominent class of safety violations that require a large number of loop iterations to surface. In C programs, buffers and strings are typically implemented using arrays. Let `i` be the variant of a loop which contains an assignment `a[i]:=`e to an array `a`. For a single iteration, we obtain

$$sp(\mathtt{a}[\mathtt{i}] := e, P) \overset{\text{def}}{=} \exists {}^{\backprime}\mathtt{a}\,.\,\mathtt{a}[\mathtt{i}] = e[\mathtt{a}/{}^{\backprime}\mathtt{a}] \wedge \forall j \neq \mathtt{i}\,.\,(\mathtt{a}[j] = {}^{\backprime}\mathtt{a}[j]) \wedge P[\mathtt{a}/{}^{\backprime}\mathtt{a}] \tag{3.5}$$

Assume further that closed forms for the variant `i` and the expression e exist (abusing our notation, we use f_e to refer to the latter). Given an initial pre-condition $P = \mathsf{T}$, we obtain the following *under-approximation* after n iterations:

$$\begin{aligned}\forall j \in [0,n)\,.\,\mathtt{a}^{\langle n\rangle}[f_{\mathtt{i}}(\mathtt{X}^{\langle 0\rangle}, j)] = f_e(\mathtt{X}^{\langle 0\rangle}, j) \wedge \\ \forall i \in \operatorname{dom}\mathtt{a}\,.\,\underbrace{\left(\exists j \in [0,n)\,.\,i = f_{\mathtt{i}}(\mathtt{X}^{\langle 0\rangle}, j)\right)}_{\text{membership test}} \vee \left(\mathtt{a}^{\langle n\rangle}[i] = \mathtt{a}^{\langle 0\rangle}[i]\right)\ ,\end{aligned} \tag{3.6}$$

where the domain ($\operatorname{dom}\mathtt{a}$) of `a` denotes the valid indices of the array. Condition (3.6) *under*-approximates the strongest post-condition, since there may exist $j_1, j_2 \in [0,n)$ such that $j_1 \neq j_2 \wedge f_{\mathtt{i}}(\mathtt{X}^{\langle 0\rangle}, j_1) = f_{\mathtt{i}}(\mathtt{X}^{\langle 0\rangle}, j_2)$ and (3.6) is unsatisfiable. A similar situation arises if a loop body π contains multiple updates of the same array.

Notably, the membership test determining whether an array element is modified or not introduces quantifier alternation, posing a challenge to contemporary decision procedures. Section 3.4 addresses the elimination of the existential quantifier.

Example 3. Consider the following loop, written in C:

```
while (i < N) {
  A[i] = i;
  i++;
}
```

The accelerator for this loop includes constraints on the new contents of the array, A', in terms of the old contents A, the initial value of i and the number of times the loop has iterated k:

$$\begin{aligned} \forall j.(\exists l.0 \leq l < k \wedge i + l = j) \Rightarrow &\quad A'[j] = j \\ \forall j.(\nexists l.0 \leq l < k \wedge j + l = j) \Rightarrow &\quad A'[j] = A[j] \\ &\quad i' = i + k \end{aligned}$$

3.3.3 Assumptions and Feasibility of Paths

The techniques discussed in Section 3.3.1 yield polynomials and constraints representing the assignment statements of π^n, but leave aside the conditional statements which determine the feasibility of the path. In the following, we demonstrate how to derive the pre-condition $\neg wlp(\pi^n, \mathsf{F})$ using the polynomials $f_{\mathtt{x}}$ for $\mathtt{x} \in \mathtt{X}$.

Let $f_{\mathtt{X}}(\mathtt{X}, n) \stackrel{\text{def}}{=} \{f_{\mathtt{x}}(\mathtt{X}, n) \,|\, \mathtt{x} \in \mathtt{X}\}$ and let $Q[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, n)]$ denote the simultaneous substitution of all free occurrences of the variables $\mathtt{x} \in \mathtt{X}$ in Q with the corresponding term $f_{\mathtt{x}}(\mathtt{X}, n)$.

Lemma 6. *The following equivalence holds:*

$$wlp(\pi^n, \mathsf{F}) \quad \equiv \quad \exists j \in [0, n) \,.\, (wlp(\pi, \mathsf{F}))\, [\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]$$

Proof. Intuitively, the path π^n is infeasible if for *any* $j < n$ the first time-frame of the suffix $\pi^{(n-j)}$ is infeasible. We prove the claim by induction over n. Due to (3.3) and (3.4) we have $f_{\mathtt{X}}(\mathtt{X}, 0) = \mathtt{X}$ and $f_{\mathtt{X}}(f_{\mathtt{X}}(\mathtt{X}, n), 1) = f_{\mathtt{X}}(\mathtt{X}, n + 1)$ (for $n \geq 0$).

Base case: $wlp(\pi, \mathsf{F}) \equiv \exists j \in [0, 0) \,.\, (wlp(\pi, \mathsf{F}))\, [\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)] = \mathsf{F}$

Induction step. We start by applying the induction hypothesis:

$$\begin{aligned} wlp(\pi^n, \mathsf{F}) &\equiv wlp(\pi, (wlp(\pi^{n-1}, \mathsf{F}))) \\ &\equiv wlp(\pi, \exists j \in [0, n-1) \,.\, (wlp(\pi, \mathsf{F}))\, [\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]) \end{aligned}$$

We consider the effect of assignments and assumptions occurring in π on the post-condition $Q \stackrel{\text{def}}{=} (\exists j \in [0, n-1) \,.\, (wlp(\pi, \mathsf{F}))\, [\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)])$ separately.

- The effect of assignments in π on Q is characterised by $Q[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X},1)]$. We obtain:

$$Q[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X},1)] \;\equiv\; \exists j \in [0,n-1)\,.\,(wlp(\pi,\mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(f_{\mathtt{X}}(\mathtt{X},1),j)] \;\equiv\; \exists j \in [1,n)\,.\,(wlp(\pi,\mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X},j)]$$

- Assumptions in π contribute the disjunct $wlp(\pi,\mathsf{F})$.

By combining both contributions into one term we obtain

$$wlp(\pi^n,\mathsf{F}) \equiv (wlp(\pi,\mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X},0)] \vee \exists j \in [1,n)\,.\,(wlp(\pi,\mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X},j)]\;,$$

which establishes the claim of Lemma 6. □

Accordingly, given a path π modifying the set of variables $\mathtt{X}$ and a corresponding set $f_{\mathtt{X}}$ of closed-form assignments, we can construct an accurate representation of π^n as follows:

$$\underbrace{[\forall j \in [0,n)\,.\,(\neg wlp(\pi,\mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X},j)]\,]}_{\text{satisfiable if } \pi^n \text{ is feasible}} \quad ; \quad \underbrace{\mathtt{X} := f_{\mathtt{X}}(\mathtt{X},n)}_{\text{assignments of } \pi^n} \tag{3.7}$$

We emphasise that our construction (unlike many acceleration techniques) does *not* restrict the assumptions in π to a limited class of relations on integers. The construction of the path (3.7), however, does require closed forms of all assignments in π. Since we do not construct closed forms for array assignments (as opposed to assignments to array indices, c.f. Section 3.3.2), we cannot apply Lemma 6 if $wlp(\pi,\mathsf{F})$ refers to an array assigned in π. In this case, we do not construct $\widehat{\pi}$.

For assignments of variables not occurring in $wlp(\pi,\mathsf{F})$, we augment the domain(s) of the variables $\mathtt{X}$ with an undefined value $\perp$ (implemented using a Boolean flag) and replace $f_{\mathtt{x}}$ with $\perp$ whenever the respective closed form is not available. Subsequently, whenever the search algorithm encounters an (abstract) counterexample, we use slicing to determine whether the feasibility of the counterexample depends on an undefined value $\perp$. If this is the case, the counterexample needs to be dismissed. Thus, any path $\widehat{\pi}$ containing references to $\perp$ is an *under-approximation* of π^n rather than an acceleration of π.

Example 4. For a path $\pi \overset{\text{def}}{=} [\mathtt{x} < 10]; \mathtt{x} := \mathtt{x}+1; \mathtt{y} := \mathtt{y}^2$, we obtain the under-approximation $\widehat{\pi} \equiv n{:=}\,*; [\forall j \in [0,n).\mathtt{x}+j < 10]; \mathtt{x} := \mathtt{x}+n; \mathtt{y} :=\perp$. A counterexample traversing $\widehat{\pi}$ is feasible if its conditions do not depend on $\mathtt{y}$.

3.3.4 Arithmetic Overflows

The fact that the techniques in Section 3.3.1 used to derive closed forms do not take arithmetic overflow into account may lead to undesired effects. For instance, the assumption made in Example 2 that the characteristic function of the predicate $(\texttt{i} + n < \texttt{BUFLEN})$ is monotonic in n does not hold in the context of bit-vectors or modular arithmetic. Since, moreover, the behaviour of arithmetic over- or under-flow in C is not specified in certain cases, we conservatively rule out all occurrences thereof in $\widehat{\pi}$. For the simple assignment $\texttt{i} := \texttt{i} + n$ in Example 2, this can be achieved by adding the assumption $(\texttt{i} + n \leq 2^l - 1)$ to $\widehat{\pi}$ (for unsigned l-bit vectors). In general, we have to add respective assumptions $(e_1 \otimes e_2 \leq 2^l - 1)$ for all arithmetic expressions $e_1 \otimes e_2$ of bit-width l and operations $\otimes$ in $\widehat{\pi}$.

While this approach is *sound* (eliminating paths from $\widehat{\pi}$ does not affect the correctness of the instrumented program, since all behaviours following an overflow are still reachable via non-approximated paths), it imposes restrictions on the range of n. Therefore, the resulting approximation $\widehat{\pi}$ deviates from the acceleration π^* of π. Unlike acceleration over linear affine relations, this adjustment makes our approach bit-level accurate. We emphasise that the benefit of the instrumentation can still be substantial, since the number of iterations required to trigger an arithmetic overflow is typically large.

3.3.5 Path Selection

In the following, we discuss heuristics to select paths π to accelerate. Depending on which model checking technique we are incorporating acceleration into, several path selection strategies are available. Some model checkers already come equipped with a strategy for enumerating paths, for example Impact [81] enumerates paths by iteratively unrolling the CFG of the program under analysis. During this process, if a path is found to be "looping" (i.e. some program location is visited repeatedly) then that path is a candidate for acceleration. By contrast, if we use a model checking technique that is not explicitly path based (such as Bounded Model Checking), we must devise a strategy for selecting paths to accelerate.

A necessary condition for π to be acceleratable is that π^2 is feasible, for if it were not, π^n (for $n > 1$) would be infeasible, resulting in a trivial accelerator. Accordingly, paths π for which π^2 is feasible are promising candidates for acceleration.

We can find such paths by using symbolic execution to build a system of constraints and solving the system with a SAT solver. Our encoding guarantees that if these constraints have a solution, the solution includes a path π where π^2 is feasible.

Let L denote the program fragment denoting the loop body. We instrument L with "distinguisher" and "shadow distinguisher" variables, which indicate which branches are taken. Concretely, for each statement $\pi_i \square \pi_j$ in L we create boolean variables d_i, s_i, d_j, and s_j. We create the instrumented program $\mathrm{Instr}(L)$ by prepending the statement d_i:= `false`, appending the statement `assume`$(d_i = s_i)$, and then replacing the statement $\pi_i \square \pi_j$ with

$$(d_i \texttt{:= true}; \pi_i) \square (d_j \texttt{:= true}; \pi_j)$$

$\mathrm{Instr}(L)$ has the property that when it has finished executing, each of the d_i will be true iff the corresponding branch was taken. Furthermore, we have $d_i = s_i$ for each i. We now sequentially compose two copies of this instrumented program:

$$\mathrm{Instr}(L); \mathrm{Instr}(L)$$

We assume that the s_i are initialised non-deterministically at the beginning of this program fragment. An example of this construction is shown in Figure 3.3.

Since each of the distinguisher variables d_i is equal to the shadow distinguisher s_i at the end of each copy of $\mathrm{Instr}(L)$, we know that the only feasible paths through this program are those in which both copies took the same path. This path is identified by the values of the s_i. So if there are *any* feasible paths through this program, they identify a path π such that π^2 in the original program is feasible. We can identify a feasible path through this program by appending the statement `assert(false)` to the end of the program and using a BMC-based model checker (which ultimately creates a SAT/SMT instance) to check the safety of the constructed program. We can iterate this process to enumerate candidate paths: if we have previously found the paths $\pi_1, \ldots, \pi_n$ we can add assumptions to the end of our path-enumerating program to prevent the rediscovery of these π_i.

3.4 Eliminating Quantifiers from Approximations

A side effect of the approximation steps in Section 3.3.2 and Section 3.3.3 is the introduction of quantified assumptions. While quantification is often unavoidable in the presence of arrays, it is a detriment to performance of the decision procedures

```
                                        | ($d_1$ := false; $d_2$ := false);
                                        | ($d_1$ := true; assume(x > 0); x := x-1)
                                        | □
                                        | ($d_2$ := true; assume(x < 0) ; x :=
                                        | x+1);
(assume(x > 0); x := x-1) □             | (assume($d_1 = s_1$); assume($d_2 = s_2$));
(assume(x < 0) ; x := x+1)              | ($d_1$ := false; $d_2$ := false);
                                        | ($d_1$ := true; assume(x > 0); x := x-1)
                                        | □
                                        | ($d_2$ := true; assume(x < 0) ; x :=
                                        | x+1);
                                        | (assume($d_1 = s_1$); assume($d_2 = s_2$))
```

Figure 3.3: A program instrumented to enumerate acceleratable paths

underlying the verification tools. In the worst case, quantifiers may result in the undecidability of path feasibility.

In the following, we discuss two techniques to eliminate or reduce the number of quantifiers in assumptions occurring in $\widehat{\pi}$.

3.4.1 Eliminating Quantifiers over Monotonic Predicates

We show that the quantifiers introduced by the technique presented in Section 3.3.3 can be eliminated if the predicate is monotonic in the quantified parameter.

Definition 7 (Representing Function, Monotonicity)**.** The representing function f_P of a predicate P with the same domain takes, for each domain value, the value 0 if the predicate holds, and 1 if the predicate evaluates to false, i.e., $P(\mathtt{X}) \Leftrightarrow f_P(\mathtt{x}) = 0$. A predicate $P(n) : \mathbb{N} \to \mathbb{B}$ is monotonically increasing (decreasing) if its representing function $f_P(n) : \mathbb{N} \to \mathbb{N}$ is monotonically increasing (decreasing), i.e., $\forall m, n \,.\, m \leq n \Rightarrow f_P(m) \leq f_P(n)$.

We extend this definition to predicates over variables X and $n \in \mathbb{N}$ as follows: $P(\mathtt{X}, n)$ is monotonically increasing in n if $(m \leq n) \wedge P(\mathtt{X}, n) \wedge \neg P(\mathtt{X}, m)$ is unsatisfiable.

Proposition 8. *$P(\mathtt{X}, n-1) \equiv \forall i \in [0, n) \,.\, P(\mathtt{X}, i)$ if P is monotonically increasing in i.*

The validity of Proposition 8 follows immediately from the definition of monotonicity. Accordingly, it is legitimate to replace universally quantified predicates in $\widehat{\pi}$ with their corresponding unquantified counterparts (c.f. Proposition 8).

$$\begin{tikzcd}
u \arrow[r, "{[P_1 \vee P_2]}"] & w \arrow[r, "\pi"] & v
\end{tikzcd}$$

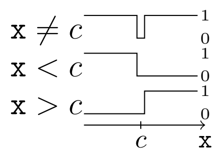
$$\begin{aligned}
&sp([P_1 \vee P_2];\pi, Q) \equiv \\
&\quad sp(\pi, (P_1 \wedge Q) \vee (P_2 \wedge Q)) \equiv \\
&\qquad sp(\pi, P_1 \wedge Q) \vee sp(\pi, P_2 \wedge Q)
\end{aligned}$$

$$\begin{tikzcd}
 & w_1 \arrow[dr, bend left, "\pi"] & \\
u \arrow[ur, bend left, "{[P_1]}"] \arrow[dr, bend right, "{[P_2]}"'] & & v \\
 & w_2 \arrow[ur, bend right, "\pi"'] &
\end{tikzcd}$$

$$\begin{aligned}
&sp(([P_1];\pi) \Box ([P_2];\pi), Q) \equiv \\
&\quad sp([P_1];\pi, Q) \vee sp([P_2];\pi, Q) \equiv \\
&\quad sp(\pi, (P_1 \wedge Q)) \vee sp(\pi, (P_2 \wedge Q))
\end{aligned}$$

Figure 3.4: Splitting disjunctive assumptions preserves program behaviour

This technique, however, fails for simple cases such as $\mathtt{x} \neq c$ (c being a constant). In certain cases, the approach can still be applied after *splitting* a non-monotonic predicate P into monotonic predicates $\{P_1, \ldots, P_m\}$ such that $P \equiv \bigvee_{i=1}^{m} P_i$ (as illustrated in the Figure to the right). Subsequently, the path π guarded by P can be split as outlined in Figure 3.4. This transformation preserves reachability (a proof for $m = 2$ is given in Figure 3.4).



This approach is akin to trace partitioning [52], however, our intent is quantifier elimination rather than refining an abstract domain. We rely on a template-based approach to identify predicates that can be split (a constraint solver-based approach is bound to fail if c is symbolic). While this technique effectively deals with a broad number of standard cases, it does fail for quantifiers over array indices, since the array access operation is not monotonic.

3.4.2 Eliminating Quantifiers in Membership Tests for Array Indices

This sub-section aims at replacing the existentially quantified membership test in Predicate (3.6) by a quantifier-free predicate. To define a set of sufficient (but not necessary) conditions for when this is possible, we introduce the notion of increasing and dense array indices (c.f. [62]):

Definition 9 (Increasing and Dense Variables)**.** A scalar variable x is (strictly) *increasing* in π^n iff $\forall j \in [0, n) \,.\, \mathtt{x}^{\langle j+1\rangle} \geq \mathtt{x}^{\langle j\rangle}$ ($\forall j \in [0, n) \,.\, \mathtt{x}^{\langle j+1\rangle} > \mathtt{x}^{\langle j\rangle}$, respectively). Moreover, an increasing variable i is *dense* iff

$$\forall j \in [0, n) \,.\, \left(\mathtt{x}^{\langle j+1\rangle} = \mathtt{x}^{\langle j\rangle}\right) \vee \left(\mathtt{x}^{\langle j+1\rangle} = \mathtt{x}^{\langle j\rangle} + 1\right) \,.$$

Variables decreasing in π^n are defined analogously. A variable is monotonic (in π^n) if it is increasing or decreasing (in π^n).

Note that if the closed form $f_{\mathtt{x}}(\mathtt{X}^{\langle 0\rangle}, n)$ of a variable $\mathtt{x}$ is a linear polynomial, then $\mathtt{x}$ is necessarily monotonic. The following proposition uses this property:

Proposition 10. *Let $f_{\mathtt{x}}(\mathtt{X}^{\langle 0\rangle}, j)$ be the closed form (3.2) of $\mathtt{x}^{\langle j\rangle}$, where $\alpha_{(2\cdot k+2)} = 0$, i.e., the polynomial $f_{\mathtt{x}}$ is linear. Then $\Delta f_{\mathtt{x}} \overset{\text{def}}{=} f_{\mathtt{x}}(\mathtt{X}^{\langle 0\rangle}, j+1) - f_{\mathtt{x}}(\mathtt{X}^{\langle 0\rangle}, j)$ (for $j \in [0, n)$) is the (symbolic) constant $\sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathtt{x}_i^{\langle 0\rangle} + \alpha_{(2\cdot k+1)}$. The variable $\mathtt{x}$ is (strictly) increasing in π^n if $\Delta f_{\mathtt{x}} \geq 0$ ($\Delta f_{\mathtt{x}} > 0$, respectively) and dense if $0 \leq \Delta f_{\mathtt{x}} \leq 1$.*

Lemma 11. *Let $f_{\mathtt{x}}(\mathtt{X}^{\langle 0\rangle}, j)$ be a linear polynomial representing the closed form (3.2) of $\mathtt{x}^{\langle j\rangle}$ (as in Proposition 10). The following logical equivalence holds:*

$$\exists j \in [0, n) \,.\, \mathtt{x} = f_{\mathtt{x}}(\mathtt{X}^{\langle 0\rangle}, j) \;\equiv \tag{3.8}$$
$$\begin{cases} ((\mathtt{x} - \mathtt{x}^{\langle 0\rangle}) \mod \Delta f_{\mathtt{x}} = 0) \wedge (\frac{\mathtt{x}-\mathtt{x}^{\langle 0\rangle}}{\Delta f_{\mathtt{x}}} < n) & \textit{if } \mathtt{x} \textit{ is strictly increasing} \\ \mathtt{x} - \mathtt{x}^{\langle 0\rangle} \leq (n-1) \cdot \Delta f_{\mathtt{x}} & \textit{if } \mathtt{x} \textit{ is dense} \\ \mathtt{x} - \mathtt{x}^{\langle 0\rangle} < n & \textit{if both of the above hold} \end{cases}$$

The validity of Lemma 11 follows immediately from Proposition 10. Lemma 11 allows us to replace the existentially quantified membership test in Predicate (3.6) by a quantifier-free predicate if one of the side conditions in (3.8) holds. Given that the path prefix reaches the entry node of a loop, these conditions $\Delta f_{\mathtt{x}} > 0$ and $0 \leq \Delta f_{\mathtt{x}} \leq 1$ can be checked using a satisfiability solver.

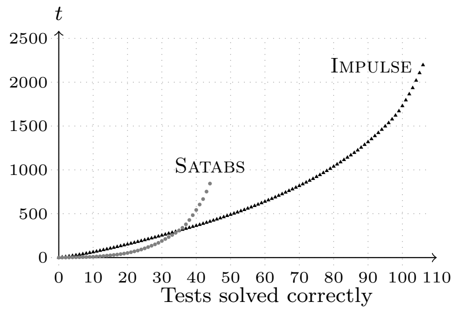
Example 5. Let $\pi \overset{\text{def}}{=} \mathtt{a}[\mathtt{x}] := \mathtt{x};\ \mathtt{x} := \mathtt{x} + \mathtt{1}$ be the body of a loop. By instantiating (3.6), we obtain the condition

$$\forall j \in [0, n) \,.\mathtt{a}[\grave{}\mathtt{x} + j] = \grave{}\mathtt{x} + j \;\wedge \forall i.\, (\exists j \in [0, n) \,.\, i = \grave{}\mathtt{x} + j) \;\vee\; (\mathtt{a}[i] = \grave{}\mathtt{a}[i])\,,$$

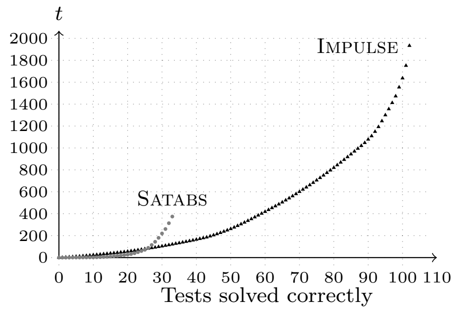
in which the existentially quantified term can be replaced by $\mathtt{x} - \grave{}\mathtt{x} < n$.

3.5 Implementation and Experimental Results

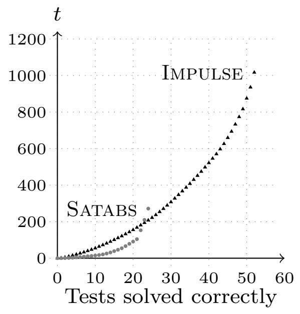
Our under-approximation technique is designed to extend existing verifiers. To demonstrate its versatility, we implemented Impulse, a tool combining under-approximation with the two popular software verification techniques lazy abstraction with interpolants (LAwI) [81] and bounded model checking (specifically, Cbmc [29] version 4.7). The underlying SMT solver used throughout was version 4.2 of Z3. Impulse comprises two phases:



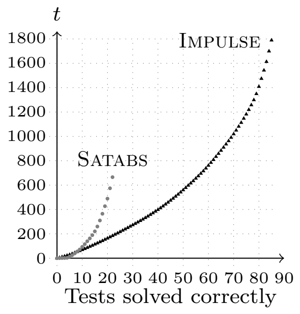
(a) Safe and unsafe, buffer size 10



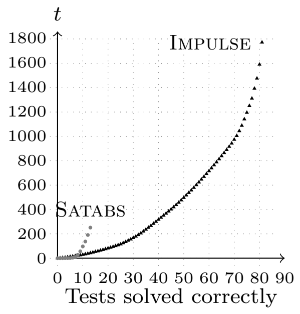
(b) Safe and unsafe, buffer size 10^2



(c) Safe/unsafe, b.-size 10^3



(d) Unsafe, buffer size 10



(e) Unsafe, buffer size 10^2

Figure 3.5: Verification run-times (cumulative) of VERISEC benchmark suite

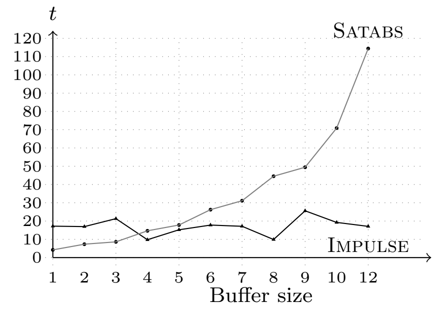
1. IMPULSE first explores the paths of the CFG following the LAWI paradigm. If IMPULSE encounters a path containing a loop with body π, it computes $\widehat{\pi}$ (processing inner loops first in the presence of nested loops), augments the CFG accordingly, and proceeds to phase 2.
2. CBMC inspects the instrumented CFG up to an iteration bound of 2. If no counterexample is found, IMPULSE returns to phase 1.

In phase 1, spurious counterexamples serve as a catalyst to refine the current approximation of safely reachable states, relying on the weakest precondition[1] to generate the required Hoare triples. Phase 2 takes advantage of the aggressive path merging performed by CBMC, enabling fast counterexample detection.

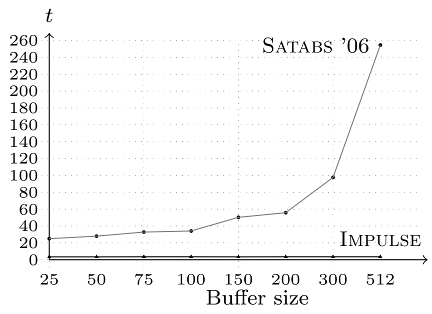
We evaluated the effectiveness of under-approximation on the VERISEC benchmark suite [71], which consists of manually sliced versions of several open source programs that contain buffer overflow vulnerabilities. Of the 284 test cases of VERISEC, 144 are labelled as containing a buffer overflow, and 140 are labelled as safe.[2] The safety violations range from simple unchecked string copy into static buffers, up

[1]In a preliminary interpolation-based implementation, Z3 was in many cases unable to provide interpolants for path formulas $\widehat{\pi}$ with quantifiers, arrays and bit-vectors.

[2]Our new technique discovered bugs in 10 of the benchmarks that had been labelled safe. SATABS timed out before identifying these bugs.



(a) Single VERISEC test, varying buffer size



(b) SATABS w. loop detect on Aeon 0.02a

Figure 3.6: Run-time dependency on buffer size for unsafe benchmarks

Table 3.1: Number of accelerated loops (in 284 programs)

Tool	**Solved**	**# loops accelerated**	**# programs accelerated**
IMPULSE (w/o acc.)	17	0	0
IMPULSE	102	258	119
SATABS	33	0	0

to complex loops with pointer arithmetic. The buffer size in each benchmark (c.f. `BUFLEN` in Figure 3.1) is adjustable and controls the depth of the counterexample. We compared our tool with SATABS (which outperforms IMPULSE w/o approximation[6]) on buffer sizes of 10, 100 and 1000, with a time limit of 300 s and a memory limit of 2 GB on an 8-core 3 GHz Xeon CPU. Figures 3.5a through 3.5c show the cumulative run-time for the whole benchmark suite, whereas Figures 3.5d and 3.5e show only unsafe program instances. Table 3.1 provides an overview of the test cases solved by IMPULSE with or without acceleration compared to the test cases solved by SATABS, including the number of loops and programs that were accelerated. Further, our static acceleration tool accelerates loops (with symbolic rather than static bounds) in 42 programs out of the 79 candidates from the 2013 software verification competition.

Finally, on the 8 safe instances from the VERISEC benchmark that IMPULSE can solve (using interpolants computed via the weakest pre-condition), under-approximation did not improve (or impair) the run-time on safe instances.

Figure 3.6 demonstrates that the time IMPULSE requires to detect a buffer overflow does not depend on the buffer size. Figure 3.6a compares SATABS and IMPULSE on a single VERISEC benchmark with a varying size parameter, showing that SATABS takes time exponential in the size of the buffer. Figure 3.6b provides a qualitative comparison of the loop-detection technique presented in [70] with IMPULSE on the Aeon 0.02a mail transfer agent. Figure 3.6b shows the run-times of SATABS'06 with loop detection as reported in [70],[3] as well as the run-times of IMPULSE on the same problem instances and buffer sizes. SATABS'06 outperforms similar model checking tools that do not feature loop-handling mechanisms [70]. However, the run-time still increases exponentially with the size of the buffer, since the technique necessitates a validation of the unwound counterexample. IMPULSE does not require such a validation step.

[3]Unfortunately, loop detection in SATABS is neither available nor maintained anymore.

Chapter 4

Proving Safety with Loop Acceleration and Trace Automata

4.1 Introduction

Software verification can be loosely divided into two themes: finding bugs and proving correctness. These two goals are often at odds with one another, and it is rare that a tool excels at both tasks. This tension is well illustrated by the results of the 2014 Software Verification Competition (SV-COMP14) [11], in which several of the best-performing tools were based on Bounded Model Checking (BMC) [13]. The BMC-based tools were able to quickly find bugs in the unsafe programs, but were unable to soundly prove safety for the remaining programs. Conversely, many of the sound tools had difficulty in detecting bugs in the unsafe programs.

The reasons for this disparity are rooted in the very nature of contemporary verification tools. Tools aiming at proof typically rely on over-approximating abstractions and refinement techniques to derive the loop invariants required (e.g., [56, 81]). For certain classes of programs, invariants can be found efficiently using templates [12] or theorem provers [62]. For unsafe programs, however, any attempt to construct a safety invariant must necessarily fail, triggering numerous futile refinement iterations before a valid counterexample is detected. Verifiers based on the BMC paradigm (such as Cbmc [29]), on the other hand, are able to efficiently detect shallow bugs, but are unable to prove safety in most cases.

The key principle of this chapter is that BMC is able to prove safety once the unwinding bound exceeds the reachability diameter of the model [13, 69]. The diameter of non-trivial programs is however in most cases unmanageably large. Furthermore, even when the diameter is small, it is often computationally expensive to determine, as the problem of computing the exact diameter is equivalent to a 2-QBF instance.

The contribution of this chapter is a technique that reduces the diameter of a program in a way that the new, smaller diameter can be computed by means of a simple satisfiability check. The technique has two steps:

1. We first identify potentially deep program paths that can be replaced by a concise single-step summary called an *accelerator* [15, 18, 44].
2. We then remove those paths subsumed by the accelerators from the program using *trace automata* [55].

The resulting program preserves the reachable states of the original program, but is often very shallow, and consequently, we can obtain a sound verification result using BMC.

This chapter is organised as follows: We present a number of motivating examples and an outline of our approach in Section 4.2. Section 4.3 describes the construction of accelerated programs and discusses the resulting reduction of the reachability diameter of the program. In Section 4.4, we introduce restricting languages and trace automata as a means to eliminate redundant transitions from accelerated programs. The experimental evaluation based on a selection of SV-COMP14 benchmarks is presented in Section 4.5. Finally, Section 4.7 briefly surveys related work.

4.2 Motivation

In this section we will discuss the differences between proving safety and finding bugs, with reference to some SV-COMP14 benchmarks, and informally demonstrate why our method is effective for both kinds of analyses.

The program in Figure 4.1, taken from the Loops category of SV-COMP14, proved challenging for many of the participating tools, with only 6 out of the 12 entrants solving it correctly. A proof of safety for this program using an abstract interpreter requires a relational domain to represent the invariant $\mathtt{x} + \mathtt{y} = \mathtt{N}$, which is often expensive.

The program in Figure 4.2 resembles the one in Figure 4.1, except for the negated assertion at the end. This example is very easy for Bounded Model Checkers, which are able to discover a bug in a single unwinding by assigning $\mathtt{N} = 1$. A slight modification, however, illustrated in Figure 4.3, increases the number of loop iterations required to trigger the bug to 10^6, exceeding the capability of even the best BMC-based verification tools.

```
unsigned N := ∗;
unsigned x := N, y := 0;
while (x > 0) {
      x := x − 1;
      y := y + 1;
}
assert (y = N);
```

Figure 4.1: Safe program

```
unsigned N = ∗;
unsigned x := N, y := 0;
while (x > 0) {
      x := x − 1;
      y := y + 1;
}
assert (y ≠ N);
```

Figure 4.2: Unsafe program

```
unsigned N := 10^6;
unsigned x := N, y := 0;
while (x > 0) {
      x := x − 1;
      y := y + 1;
}
assert (y ≠ N);
```

Figure 4.3: "Deep" bug

```
unsigned i := ∗;             }
assume (i > 0)               } iteration counter
assume(x > 0);               } feasibility check
x := x−i;                    }
y := y+i;                    } acceleration
assume(¬underflow (x));      } iteration bound
```

Figure 4.4: Accelerated loop body

```
unsigned N := 10^6, x := N, y := 0;
while (x > 0) {
      if (∗) {
            i := ∗; assume (i > 0);
            x := x−i; y = y+i;
            assume (x ≥ 0);
      } else {
            x := x − 1; y := y + 1;
      }
}
assert (y ≠ N);
```

Figure 4.5: Accelerated unsafe program

The relative simplicity of the program statements in Figures 4.1 to 4.3 makes them amenable to *acceleration* [15, 18, 44], a technique used to compute the effect of the repeated iteration of statements over integer linear arithmetic. Specifically, the effect of i loop iterations is that x is decreased and y is increased by i. Acceleration, however, is typically restricted to programs over fragments of linear arithmetic for which the transitive closure is effectively computable, thus restricting its applicability to programs whose semantics can be soundly modelled using unbounded integers. In reality, however, the scalar variables in Figures 4.1 to 4.3 take their values from the bounded subset $\{0, \ldots, (2^{32} - 1)\}$ of the positive integers $\mathbb{N}_0$. Traditional acceleration techniques do not account for integer overflows. To address this problem, we previously introduced *under-approximate acceleration*, bounding the acceleration to the interval in which the statements behave uniformly [67].

The code snippet in Figure 4.4 represents an under-approximating accelerator for the loop bodies in Figures 4.1, 4.2, and 4.3. We introduce an auxiliary variable i representing a non-deterministic number of loop iterations. The subsequent assumption

```
unsigned N := 10^6, x := N, y := 0;
if (x > 0) {
    x := x − 1; y := y + 1;
    if (x > 0) {
        x := x − 1; y := y + 1;
        if (x > 0) {
            x := x − 1;
            y := y + 1;
            assert (x ≤ 0);
        }
    }
}
assert (y = N);
```

Figure 4.6: Unwinding ($k = 3$) of safe program with $\mathtt{N} = 10^6$

```
    unsigned N := ∗, x := N, y := 0;
    bool g := ∗;
1:  while (x > 0) {
        if (∗) {
            assume (¬g);
2:          i := ∗; x := x−i; y = y+i;
            assume (x ≥ 0);
3:          g := T;
        } else {
            x := x − 1; y := y + 1;
            assume (underflow (x));
            g := F;
        }
    }
4:  assert (y = N);
```

Figure 4.7: Accelerated and instrumented safe program

guarantees that the accelerated code reflects at least one iteration (and is optional in this example). The assumption that follows warrants the feasibility of the accelerated trace (in general, this condition may contain quantifiers [67]). The effect of i iterations is encoded using the two assignment statements, which constitute the closed forms of the recurrence relations corresponding to the original assignments. The final assumption guarantees that i lies in the range in which the right-hand sides of the assignments behave linearly.

In general, under-approximating accelerators do not reflect all feasible iterations of the loop body. Accordingly, we cannot simply replace the original loop body. Instead, we add back the accelerator as an additional path through the loop, as illustrated in Figure 4.5.

The transformation preserves safety properties—that is to say, an accelerated program has a reachable, failing assertion iff the original program does. We can see that the failing assertion in Figure 4.5 is reachable after a single iteration of the loop, by simply choosing $i = \mathtt{N}$. Since the accelerated program contains a feasible trace leading to a failed assertion, we can conclude that the original program does as well, despite having only considered a single trace of length 1.

While the primary application of BMC is bug detection, contemporary Bounded Model Checkers such as Cbmc are able to prove safety in some cases. Cbmc unwinds loops up to a predetermined bound k (see Figure 4.6). *Unwinding assertions* are one possible mechanism to determine whether further unwinding is required [29,42]. The assertion ($\texttt{x} \leq \texttt{0}$) in Figure 4.6 fails if there are feasible program executions traversing the loop more than three times. It is obvious that this assertion will fail for any $k < 10^6$.

Unfortunately, acceleration is ineffective in this setting. Since the accelerator in Figure 4.5 admits $i = 1$, we have to consider 10^6 unwindings before we can establish the safety of the program in Figure 4.1 with $\texttt{N} = \texttt{10}^6$. For a non-deterministically assigned N, this number increases to 2^{32}.

This outcome is disappointing, since the repeated iteration of the accelerated loop body is redundant. Furthermore, there is no point in taking the unaccelerated path through the loop (unless there is an impending overflow—which can be ruled out in the given program), since the accelerator *subsumes* this execution (with $i = 1$). Thus, if we eliminate all executions that meet either of the criteria above, we do not alter the semantics of the program but may reduce the difficulty of our problem considerably.

Figure 4.7 shows an accelerated version of the safe program of Figure 4.1, but instrumented to remove redundant traces. This is achieved by introducing an auxiliary variable g which determines whether the accelerator was traversed in the previous iteration of the loop. This flag is reset in the non-accelerated branch, which, however, in our example is never feasible. It is worth noting that every feasible trace through Listing 4.1 has a corresponding feasible trace through Listing 4.7, and vice versa.

The figure to the right shows an execution of the program in Figure 4.7: This trace is both feasible and safe—the assertion on line 4 is not violated. It is not too difficult to see that *every* feasible trace through the program in Figure 4.7 has the same length, which means that we can soundly reason about its safety considering traces with a single iteration of the loop, which is a tractable (and indeed, easy) problem.

Loc.	N	x	y	i	g
1	10^4	10^4	0	0	F
2	10^4	10^4	0	0	F
3	10^4	0	10^4	10^4	F
1	10^4	0	10^4	10^4	T
4	10^4	0	10^4	10^4	T

Since the accelerated and instrumented program in Figure 4.7 is safe, we can conclude that the original program in Figure 4.1 is safe as well.

We emphasise that our approach neither introduces an over-approximation, nor requires the explicit computation of a fixed point. In addition, it is not restricted to linear integer arithmetic and bit-vectors: our prior work can generate some non-linear accelerators and also allows for the acceleration of a limited class of programs with arrays [67].

4.3 Diameter Reduction via Acceleration

In this section, we introduce a reachability-preserving program transformation that reduces the reachability diameter of a CFA. While a similar transformation is used in [67] to detect counterexamples with loops, our goal here is to reduce the diameter in order to enable safety proofs (see Section 4.4).

Definition 12 (Accelerated CFA)**.** Let $P \overset{\text{def}}{=} \langle V, E, v_0\rangle$ be a CFA over the alphabet Stmts_P, and let $\pi_1, \ldots, \pi_k$ be traces in P looping with heads $v_1, \ldots, v_k \in V$, respectively. Let $\widehat{\pi}_1, \ldots \widehat{\pi}_k$ be the (potentially under-approximating) accelerators for $\pi_1, \ldots, \pi_k$. Then the *accelerated CFA* $\widehat{P} \overset{\text{def}}{=} \langle \widehat{V}, \widehat{E}, v_0\rangle$ for P is the CFA P augmented with non-branching paths $v_i \xrightarrow{\widehat{\pi}_i} v_i$ $(1 \le i \le k)$.

A trace is *accelerated* if it traverses a path in $\widehat{P}$ that corresponds to an accelerator. A trace π_1 *subsumes* a trace π_2, denoted by $\pi_2 \preceq \pi_1$, if $[\![\pi_2]\!] \subseteq [\![\pi_1]\!]$. Accordingly, $\pi \preceq \widehat{\pi}$ and $\widetilde{\pi} \preceq \widehat{\pi}$ (by Definition 3). We extend the relation $\preceq$ to sets of traces: $\Pi_1 \preceq \Pi_2$ if $\left(\bigcup_{\pi\in\Pi_1}[\![\pi]\!]\right) \preceq \left(\bigcup_{\pi\in\Pi_2}[\![\pi]\!]\right)$. A trace π is *redundant* if $\{\pi\}$ is subsumed by a set $\Pi \setminus \{\pi\}$ of other traces in the CFA.

Lemma 13. *Let $\widetilde{\pi}$ be an under-approximating accelerator for the looping trace π. Then $\widetilde{\pi} \cdot \widetilde{\pi} \preceq \widetilde{\pi}$ holds.*

The following theorem states that the transformation in Definition 12 preserves the reachability of states and never increases the reachability diameter.

Theorem 14. *Let P be a CFA and $\widehat{P}$ a corresponding accelerated CFA as in Definition 12. Then the following claims hold:*

1. *Every trace in P is subsumed by at least one trace in $\widehat{P}$.*
2. *Let π_1 be an accelerated trace accepted by $\widehat{P}$, and let $\langle\sigma_0, \sigma\rangle \in [\![\pi_1]\!]$. Then there exists a trace π_2 accepted by P such that $\langle\sigma_0, \sigma\rangle \in [\![\pi_2]\!]$.*

Proof. Part 1 of the theorem holds because P is a sub-graph of $\widehat{P}$. For the second part, assume that $\widehat{\pi}_1, \ldots \widehat{\pi}_k$ are the accelerators occurring in π_1. Then there are $i_1, \ldots, i_k \in \mathbb{N}$ such that $\pi_2 \overset{\text{def}}{=} \pi_1[\pi_1^{i_1}/\widehat{\pi}_1]\cdots[\pi_k^{i_k}/\widehat{\pi}_k]$ and $\langle\sigma_0, \sigma\rangle \in [\![\pi_2]\!]$. $\square$

The diameter of a CFA is determined by the longest of the shortest traces from the initial state σ_0 to all reachable states [69]. Accordingly, the transformation in Definition 12 results in a reduction of the diameter if it introduces a shorter accelerated trace that results in the redundancy of this longest shortest trace. In particular, acceleration may reduce an infinite diameter to a finite one.

4.4 Checking Safety with Trace Automata

Bounded Model Checking owes its industrial success largely to its effectiveness as a bug-finding technique. Nonetheless, BMC can also be used to prove safety properties if the unwinding bound exceeds the reachability diameter. In practice, however, the diameter can rarely be determined statically. Instead, *unwinding assertions* are used to detect looping traces that become infeasible if expanded further [29]. Specifically, an unwinding assertion is a condition that fails for an unwinding bound k and a trace $\pi_1 \cdot \pi_2^k$ if $\pi_1 \cdot \pi_2^{k+1}$ is feasible, indicating that further iterations may be required to exhaustively explore the state space.

In the presence of accelerators, however, unwinding assertions are inefficient. Since $\widehat{\pi} \cdot \widehat{\pi} \preceq \widehat{\pi}$ (Lemma 13), repeated iterations of accelerators are redundant. The unwinding assertion for $\pi_1 \cdot \widehat{\pi}_2$, however, fails if $\pi_1 \cdot \widehat{\pi}_2 \cdot \widehat{\pi}_2$ is feasible. Accordingly, the approximate diameter as determined by means of unwinding assertions for an accelerated program $\widehat{P}$ is the *same* as for the corresponding non-accelerated program P.

In the following, we present a technique that remedies the deficiency of unwinding assertions in the presence of accelerators by *restricting* the language accepted by a CFA.

Definition 15 (Restriction Language)**.** Let $\widehat{P}$ an accelerated CFA for P over the vocabulary $\mathsf{Stmts}_{\widehat{P}}$. For each accelerator $\widehat{\pi} \in \mathsf{Stmts}^+_{\widehat{P}}$, let $\pi \in \mathsf{Stmts}^+_P$ be the corresponding looping trace. The *restriction language* $\mathcal{L}_R$ for $\widehat{P}$ comprises all traces with a sub-trace characterised by the regular expression $(\pi \,|\, (\widehat{\pi} \cdot \widehat{\pi}))$ for all accelerators $\widehat{\pi}$ in $\widehat{P}$ with $\pi \preceq \widehat{\pi}$.

The following lemma enables us to eliminate traces of an accelerated CFA $\widehat{P}$ that are in the restriction language $\mathcal{L}_R$.

Lemma 16. *Let $\widehat{P}$ be an accelerated CFA, and $\mathcal{L}_R$ be the corresponding restriction language. Let π_1 be a trace accepted by $\widehat{P}$ such that $\pi_1 \in \mathcal{L}_R$. Then there exists a trace π_2 which is accepted by $\widehat{P}$ such that $\pi_1 \preceq \pi_2$ and π_1 is not a sub-trace of π_2.*

Proof. The regular expression $(\pi \,|\, (\widehat{\pi} \cdot \widehat{\pi}))$ can match the trace π_1 for two reasons:

(a) The trace π_1 contains a sub-trace which is a looping trace π with a corresponding accelerator $\widehat{\pi}$ and $\pi \preceq \widehat{\pi}$. We obtain π_2 by replacing π with $\widehat{\pi}$.

(b) The trace π_1 contains the sub-trace $\widehat{\pi} \cdot \widehat{\pi}$ for some accelerator $\widehat{\pi}$. Since $\widehat{\pi} \cdot \widehat{\pi} \preceq \widehat{\pi}$ (Lemma 13), we replace the sub-trace with $\widehat{\pi}$ to obtain π_2.

Since the accelerator $\widehat{\pi}$ differs from the sub-trace it replaces in case (a), and $|\pi_2| < |\pi_1|$ in case (b), π_1 can not be contained in π_2. $\square$

Using Lemma 16 and induction over the number of traces and accelerators, it is admissible to eliminate all traces accepted by $\widehat{P}$ and contained in $\mathcal{L}_R$ without affecting the reachability of states:

Theorem 17. *Let $\mathcal{L}_{\widehat{P}}$ be the language comprising all traces accepted by an accelerated CFA $\widehat{P}$ and $\mathcal{L}_R$ be the corresponding restriction language. Then every trace $\pi \in \mathcal{L}_{\widehat{P}}$ is subsumed by the traces in $\mathcal{L}_{\widehat{P}} \setminus \mathcal{L}_R$.*

Notably, Definition 15 explicitly excludes accelerators $\widehat{\pi}$ that do not satisfy $\pi \preceq \widehat{\pi}$, a requirement that is therefore implicitly present in Lemma 16 as well as Theorem 17. The rationale behind this restriction is that strictly under-approximating accelerators $\widetilde{\pi}$ do not necessarily have this property. However, even if $\widetilde{\pi}$ does not subsume π in general, we can characterize the set of starting states in which it does:

$$\{\sigma \mid \langle\sigma, \sigma'\rangle \in [\![\pi]\!] \Rightarrow \langle\sigma, \sigma'\rangle \in [\![\widetilde{\pi}]\!]\} \tag{4.1}$$

In order to determine whether a looping path π is redundant, we presume for each accelerated looping trace π the existence of a predicate $\varphi_\pi \in \mathsf{Exprs}$ and an assumption statement $\tau_\pi \overset{\text{def}}{=} [\varphi_\pi]$ such that

$$[\![\,\tau_\pi\,]\!] \overset{\text{def}}{=} \{\langle\sigma, \sigma\rangle \mid \langle\sigma, \sigma'\rangle \in [\![\pi]\!] \Rightarrow \langle\sigma, \sigma'\rangle \in [\![\widetilde{\pi}]\!]\} \tag{4.2}$$

Analogously, we can define the dual statement $\overline{\tau}_\pi \overset{\text{def}}{=} [\neg\varphi_\pi]$. Though both $[\![\tau_\pi]\!]$ and $[\![\overline{\tau}_\pi]\!]$ are non-total transition relations, their combination $[\![\tau_\pi]\!] \cup [\![\overline{\tau}_\pi]\!]$ is total. Moreover, it does not modify the state, i.e., $[\![\tau_\pi]\!] \cup [\![\overline{\tau}_\pi]\!] \equiv [\![\mathsf{skip}]\!]$. It is therefore evident that replacing the head v of a looping trace π with the sub-graph $\begin{tikzcd} u \arrow[r, bend left, "\tau_\pi"] \arrow[r, bend right, "\overline{\tau}_\pi"'] & w \end{tikzcd}$ (and reconnecting the incoming and outgoing edges of v to u and w, respectively) preserves the reachability of states. It does, however change the traces of the CFA. After the modification, the looping traces $\tau_\pi \cdot \pi$ and $\overline{\tau}_\pi \cdot \pi$ replace π. By definition of τ_π, we have $\tau_\pi \cdot \pi \preceq \widetilde{\pi}$. Consequently, if we accelerate the newly introduced looping trace $\tau_\pi \cdot \pi$, Definition 15 and therefore Lemma 16 as well as Theorem 17 apply.

Example 6. An under-approximating accelerator for the statement $\mathtt{x} := \mathtt{x} + \mathtt{1}$, where $\mathtt{x}$ is a 32-bit-wide unsigned integer, can be given as

$$\widetilde{\pi} \overset{\text{def}}{=} i := *;\ [\mathtt{x} + i < 2^{32}]; \mathtt{x} := \mathtt{x} + i$$

with transition relation $\exists i\,.\,(\mathtt{x} + i < 2^{32}) \wedge (\mathtt{x}' = \mathtt{x} + i)$.

The discriminating statement $\overline{\tau}_\pi$ for the path from Example 6, for instance, detects the presence of an overflow. For this specific example, $\overline{\tau}_\pi$ is the assumption $[\mathtt{x} = 2^{32} - 1]$. In practice, however, the bit-level-accurate encoding of CBMC provides a mechanism to detect an overflow *after* it happened. Therefore, we introduce statements $\overline{\tau}_\pi \overset{\text{def}}{=} [\mathsf{overflow}(\mathtt{x})]$ and $\tau_\pi \overset{\text{def}}{=} [\neg\mathsf{overflow}(\mathtt{x})]$ that determine the presence of an overflow at the end of the looping trace. The modification and correctness argument for this construction is analogous to the one above.

In order to recognize redundant traces, we use a *trace automaton* that accepts the restriction language $\mathcal{L}_R$.

Definition 18 (Trace Automaton)**.** A trace automaton T_R for $\mathcal{L}_R$ is a deterministic finite automaton (DFA) over the alphabet $\mathsf{Stmts}_{\widehat{P}}$ that accepts $\mathcal{L}_R$.

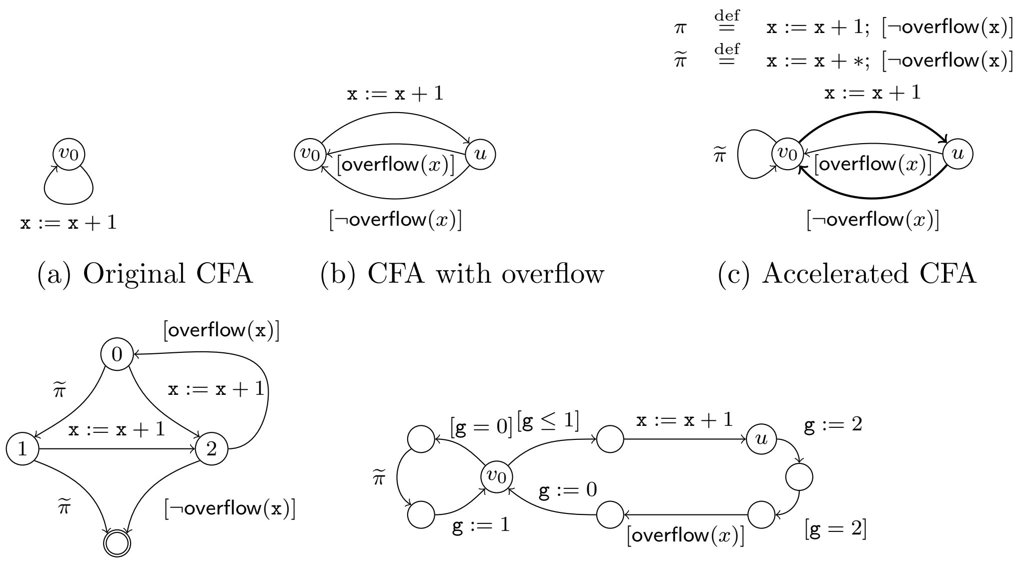
Since $\mathcal{L}_R$ is regular, so is its complement $\overline{\mathcal{L}}_R$. In the following, we describe an instrumentation of a CFA $\widehat{P}$ which guarantees that every trace accepted by T_R and $\widehat{P}$ becomes infeasible. To this end, we construct a DFA T_R recognising $\mathcal{L}_R$, starting out with an ϵ-NFA which we then determinise using the subset construction [1]. While this yields (for a CFA with k statements) a DFA with $O(2^k)$ states in the worst case, in practice the DFAs generated are much smaller.

We initialise the set the vertices of the instrumented CFA $\tilde{P}$ to the vertices of $\widehat{P}$. We inline T_R by creating a fresh integer variable g in $\tilde{P}$ which encodes the state of T_R and is initialised to 0. For each edge $u \xrightarrow{s} v \in \widehat{P}$, we consider all transitions $n \xrightarrow{s} m \in T_R$. If there are no such transitions, we copy the edge $u \xrightarrow{s} v$ into $\tilde{P}$. Otherwise, we add edges as follows:

- If m is an accepting state, we do not add an edge to $\tilde{P}$.
- Otherwise, construct a new statement $l \overset{\text{def}}{=} [\mathsf{g} = n]; \mathsf{g} := m;\ s$ and add the path $u \xrightarrow{l} v$ to $\tilde{P}$, which simulates the transition $n \xrightarrow{s} m$.

Since we add at most one edge to $\tilde{P}$ for each transition in T_R, this construction's time and space complexity are both $\Theta(|\widehat{P}| + |T_R|)$. By construction, if a trace π accepted by CFA $\tilde{P}$ projected to $\mathsf{Stmts}_{\widehat{P}}$ is contained in the restriction language $\mathcal{L}_R$, then π is infeasible. Conceptually, our construction suppresses traces accepted by $\mathcal{L}_R$ and retains the remaining executions.

An example is given in Figure 4.8. The CFA in Figure 4.8a represents an unaccelerated loop with a single path through its body. After adding an extra path to account for integer overflow, we arrive at the CFA in Figure 4.8b. We are able to find an accelerator for the non-overflowing path, which we add to the CFA resulting



(a) Original CFA (b) CFA with overflow (c) Accelerated CFA

(d) Trace automaton (e) Restricted Accelerated CFA

Figure 4.8: Accelerating a looping path

in Figure 4.8c. We use $\widetilde{\pi}$ to represent the accelerator π for the corresponding path. Then the restriction language is represented by the regular expression $(\pi \mid \widetilde{\pi} \cdot \widetilde{\pi})$. The corresponding 4-state trace automaton is shown in Figure 4.8d. By combining the trace automaton and the CFA we obtain the restricted CFA in Figure 4.8e (after equivalent paths have been collapsed).

In the restricted CFA $\tilde{P}$, looping traces π that can be accelerated and redundant iterations of accelerators are infeasible and therefore do not trigger the failure of unwinding assertions. A CFA is safe if all unwinding assertions hold and no safety violation can be detected for a given bound k. The reduction of the diameter achieved by acceleration (Section 4.3) in combination with the construction presented in this section enables us to establish the safety of CFAs in cases in which traditional BMC would have been unable to do so. Section 4.5 provides an experimental evaluation demonstrating the viability of our approach.

4.5 Experimental Evaluation

We evaluate the effect of instrumenting accelerated programs with trace automata and determine the direct cost of constructing the automata as well as the impact of trace automata on the ability to find bugs on the one hand and prove safety on the other.

Table 4.1: Summary of experimental results

	#Benchmarks	CBMC		#Benchmarks accelerated	CBMC + Acceleration			CBMC + Acceleration + Trace Automata		
		#Correct	Time(s)		#Correct	Acceleration Time (s)	Checking time (s)	#Correct	Acceleration Time (s)	Checking Time (s)
SV-COMP14 safe	35	14	298.73	21	2	23.24	244.72	14	23.86	189.61
SV-COMP14 unsafe	32	20	394.96	18	11	15.79	197.94	12	16.51	173.74
Crafted safe	15	0	11.42	15	0	2.75	32.41	15	2.91	1.59
Crafted unsafe	14	0	9.03	14	14	2.85	12.24	14	2.95	2.55

Our evaluation is based on the LOOPS category of the benchmarks from SV-COMP14 and a number of small but difficult hand-crafted examples. Our hand-crafted examples require precise reasoning about arithmetic and arrays. The unsafe examples have deep bugs, and the safe examples feature unbounded loops. The SV-COMP14 benchmarks are largely arithmetic in nature. They often require non-trivial arithmetic invariants to be inferred, but rarely require complex reasoning about arrays. Furthermore, all bugs of the unsafe SV-COMP14 benchmarks occur within a small number of loop iterations.

In all of our experiments we used CBMC taken from the public SVN at r3849 to perform the transformation. Since CBMC's acceleration procedure generates assertions with quantified arrays, we used Z3 [38] version 4.3.1 as the backend decision procedure. All of the experiments were performed with a timeout of 30 s and very low unwinding limits. We used an unwinding limit of 100 for unaccelerated programs and an unwinding limit of 3 for their accelerated counterparts.

The version of CBMC we use has incomplete acceleration support, e.g., it is unable to accelerate nested loops. As a result, there are numerous benchmarks that it cannot accelerate. We stress that our goal here is to evaluate the effect of adding trace automata to accelerated programs. Acceleration has already proven to be a useful technique for both bug-finding and proof [58, 67, 70, 90, 91] and we are interested in how well inlined trace automata can complement it.

Our experimental results are summarised in Table 4.1, and the full results are given in Section 4.6. We discuss the results in the remainder of this section.

Cost of Trace Automata. To evaluate the direct cost of constructing the trace automata, we direct the reader's attention to Table 4.1 and the columns headed "acceleration time". The first "acceleration time" column shows how long it took to generate an accelerated program without a trace automaton, whereas the second shows how long it took when a trace automaton was included. For all of these benchmarks, the additional time taken to build and insert the trace automaton is negligible. The "size increase" column in Tables 4.2, 4.3, and 4.4 in Section 4.6 shows how much larger the instrumented binary is than the accelerated binary, expressed as a percentage of the accelerated binary's size. The average increase is about 15%, but the maximum increase is 77%. There is still room for optimisation, as we do not minimise the automata before inserting them.

Bug Finding. In the following, we evaluate the effectiveness of our technique for bug finding. The current state-of-the-art method for bug finding is BMC [11]. To provide a baseline for bug finding power, we start by evaluating the effect of just combining acceleration with BMC. We then evaluate the impact of adding trace automata, as compared to acceleration without trace automata. Our hypothesis is that adding trace automata has negligible impact on acceleration's ability to find bugs. The statistics we use to measure these effects are the number of bugs found and the time to find them. We measure these statistics for each of three techniques: BMC alone, acceleration with BMC, and our combination of acceleration, trace automata and BMC.

The results are summarised in Table 4.1. In SV-COMP14, almost all of the bugs occur after a small number of unwindings. In these cases, there are no deep loops to accelerate so just using CBMC allows the same bugs to be reached, but without the overhead of acceleration (which causes some timeouts to be hit). In the crafted set the bugs are much deeper, and we can see the effect of acceleration in discovering these bugs – none of the bugs are discovered by CBMC, but each of the configurations using acceleration finds all 14 bugs.

In both of the benchmark sets, adding trace automata does not negatively impact the bug finding ability of acceleration. Indeed, for the crafted set the addition of trace automata significantly improves bug finding performance – the total time needed to find the 14 bugs is reduced from 12.31s to 1.85s.

Safety Proving. We evaluate the effectiveness of our technique for proving safety, the key contribution of this chapter. Our two benchmark sets have very different

```
unsigned N := *, i;
int a[M], b[M], c[M]
for (i = 0; i < M; i := i + 1) {
    c[i] := a[i] + b[i];
}
for (i = 0; i < M; i := i + 1) {
    assert (c[i] = a[i] + b[i]);
}
```

Figure 4.9: The SUM_ARRAYS benchmark from SV-COMP14

characteristics with respect to the safety proofs required for their safe examples. As can be seen from Table 4.1, 14 of the SV-COMP14 benchmarks can be proved safe using just BMC. That is, they can be exhaustively proved safe after a small number of loop unwindings. For the 14 cases that were provable using just BMC, none had loops that could execute for more than 10 iterations.

Of the 35 safe SV-COMP14 benchmarks, 21 contained loops that could be accelerated. Of these 21 cases, 14 were proved safe using trace automata. These are not the same 14 cases that were proved by CBMC, and notably 8 cases with unbounded loops are included, which would be impossible to prove safe with just BMC. Additionally we were able to solve the SUM_ARRAY_TRUE benchmark (shown in Fig. 4.9) in 1.75s. Of all the tools entered in SV-COMP14, the only tools to claim "safe" for this benchmark were BMC-based, and as such do not generate safety proofs.

For the 7 cases where accelerators were produced but we were unable to prove safety, 5 are due to timeouts, 1 is a crash in CBMC and 1 is an "incomplete". The 5 timeouts are due to the complexity of the SMT queries we produce. For these timeout cases, we generate assertions which contain non-linear multiplication and quantification over arrays, which are very difficult for Z3 to solve. The "incomplete" case (TREX03_TRUE) requires reasoning about accelerated paths that commute with each other, which we leave as future work.

4.6 Detailed Experimental Results

Tables 4.2, 4.3, and 4.4 show the detailed experimental results for Table 4.1 in Section 4.5.

4.7 Related Work

The diameter of a transition system was introduced in Biere et al.'s seminal paper on BMC [13] in the context of finite-state transition relations. For finite-state transition relations, approximations of the diameter can be computed symbolically by constraining the unwound transition relation to exclude executions that visit states repeatedly [69]. For software, however, this technique is ineffective. Baumgartner and Kühlmann use structural transformations of hardware designs to reduce the reachability diameter of a hardware design to obtain a complete BMC-based verification method [6]. This technique is not applicable in our context.

Trace automata are introduced in [55] as abstractions of safe traces of CFAs [56], constructed by means of interpolation. We use trace automata to recognize redundant traces.

Acceleration amounts to computing the transitive closure of a infinite state transition relation [15, 18, 44]. Acceleration has been successfully combined with abstract interpretation [90] as well as interpolation-based invariant construction [58]. These techniques rely on over-approximate abstractions to prove safety. We previously used acceleration and under-approximation to quickly find deep bugs [67, 70]. The quantified transition relations used to encode under-approximations pose an insurmountable challenge to interpolation-based refinement techniques [67], making it difficult to combine the approach with traditional software model checkers.

Name	Expected	CBMC		Accelerated?	CBMC + Acceleration			CBMC + Acceleration + Trace Automata			
		Result	Time(s)		Result	Acceleration time (s)	Checking time (s)	Result	Acceleration time (s)	Checking time (s)	Size increase
SV-COMP14											
array_true.c	✔	✔	0.04s		—	—	—	—	—	—	—
bubble_sort_true.c	✔	T/O	30.00s	Yes	T/O	3.90s	30.00s	T/O	3.97s	30.00s	20%
count_up_down_true.c	✔	?	0.84s	Yes	?	0.20s	1.20s	✔	0.21s	0.25s	11%
eureka_01_true.c	✔	✔	12.64s	Yes	T/O	1.90s	30.00s	T/O	1.95s	30.00s	34%
eureka_05_true.c	✔	✔	0.11s	Yes	↯	0.56s	2.64s	✔	0.57s	2.22s	31%
for_infinite_loop_1_true.c	✔	?	0.05s	Yes	?	0.12s	0.09s	✔	0.13s	0.06s	11%
for_infinite_loop_2_true.c	✔	✘	0.08s	Yes	✘	0.13s	0.05s	✔	0.14s	0.07s	12%
heavy_true.c	✔	T/O	30.00s		—	—	—	—	—	—	—
insertion_sort_true.c	✔	T/O	30.00s	Yes	↯	0.46s	30.00s	↯	0.48s	30.00s	18%
invert_string_true.c	✔	✔	0.12s	Yes	↯	0.87s	30.00s	✔	0.92s	2.13s	28%
linear_sea.ch_true.c	✔	?	3.78s	Yes	T/O	0.33s	30.00s	✔	0.35s	0.26s	20%
lu.cmp_true.c	✔	✔	0.34s		—	—	—	—	—	—	—
matrix_true.c	✔	✔	0.03s		—	—	—	—	—	—	—
n.c11_true.c	✔	?	0.91s		—	—	—	—	—	—	—
n.c24_true.c	✔	T/O	30.00s	Yes	↯	3.60s	11.41s	T/O	3.66s	30.00s	17%
n.c40_true.c	✔	✔	0.04s	Yes	✔	0.25s	0.14s	✔	0.26s	0.15s	11%
nec40_true.c	✔	✔	0.04s	Yes	✔	0.25s	0.13s	✔	0.25s	0.17s	11%
string_true.c	✔	✔	11.20s		—	—	—	—	—	—	—
sum01_true.c	✔	?	0.81s	Yes	?	0.50s	6.24s	✔	0.51s	0.43s	19%
sum03_true.c	✔	?	0.07s	Yes	?	0.47s	0.23s	✔	0.46s	0.22s	17%
sum04_true.c	✔	✔	0.00s	Yes	?	0.23s	0.22s	✔	0.24s	0.13s	11%
sum_array_true.c	✔	↯	30.00s	Yes	↯	0.56s	30.00s	✔	0.62s	1.75s	29%
terminator_02_true.c	✔	✔	2.58s		—	—	—	—	—	—	—
terminator_03_true.c	✔	T/O	30.00s		—	—	—	—	—	—	—
trex01_true.c	✔	?	13.96s		—	—	—	—	—	—	—
trex02_true.c	✔	?	1.27s		—	—	—	—	—	—	—
trex03_true.c	✔	?	9.51s	Yes	?	6.22s	0.75s	?	6.09s	1.69s	54%
trex04_true.c	✔	?	0.91s		—	—	—	—	—	—	—
veris.c_NetBSD-libc__loop_true.c	✔	✔	17.61s		—	—	—	—	—	—	—
veris.c_OpenSER__cases1_stripFullBoth_arr_true.c	✔	T/O	30.00s	Yes	?	1.05s	11.58s	T/O	1.16s	30.00s	77%
veris.c_sendmail__tTflag_arr_one_loop_true.c	✔	✔	0.88s		—	—	—	—	—	—	—
vogal_true.c	✔	✔	10.75s	Yes	T/O	1.60s	30.00s	T/O	1.85s	30.00s	64%
while_infinite_loop_1_true.c	✔	?	0.03s	Yes	?	0.01s	0.02s	✔	0.01s	0.03s	15%
while_infinite_loop_2_true.c	✔	?	0.06s	Yes	?	0.03s	0.02s	✔	0.03s	0.05s	16%
while_infinite_loop_3_true.c	✔	?	0.07s		—	—	—	—	—	—	—
Total	35	14	298.73s	21	2	23.24s	244.72s	14	23.86s	189.61s	

Key: Safe: ✔, Unsafe: ✘, Timeout: T/O, Crash: ↯, Incomplete (unable to prove safety or find a bug): ?

Table 4.2: Detailed experimental results for safe SV-COMP benchmarks

		CBMC		Accelerated?	CBMC + Acceleration			CBMC + Acceleration + Trace Automata			
Name	Expected	Result	Time(s)		Result	Acceleration time (s)	Checking time (s)	Result	Acceleration time (s)	Checking time (s)	Size increase
SV-COMP14											
array_false.c	✘	✘	0.03s		—	—	—	—	—	—	—
bubble_sort_false.c	✘	T/O	30.00s		—	—	—	—	—	—	—
compact_false.c	✘	T/O	30.00s		—	—	—	—	—	—	—
count_up_down_false.c	✘	✘	0.26s	Yes	✘	0.20s	0.22s	✘	0.21s	0.30s	11%
eureka_01_false.c	✘	T/O	30.00s	Yes	T/O	1.98s	30.00s	T/O	2.00s	30.00s	27%
for_bounded_loop1_false.c	✘	✘	0.67s		—	—	—	—	—	—	—
heavy_false.c	✘	T/O	30.00s		—	—	—	—	—	—	—
insertion_sort_false.c	✘	T/O	30.00s	Yes	↯	0.64s	12.31s	↯	0.62s	14.58s	17%
invert_string_false.c	✘	T/O	30.00s	Yes	↯	0.61s	30.00s	✘	0.64s	3.00s	17%
linear_search_false.c	✘	✘	0.47s	Yes	✘	0.36s	0.18s	✘	0.38s	0.34s	20%
ludcmp_false.c	✘	✘	0.45s		—	—	—	—	—	—	—
matrix_false.c	✘	T/O	30.00s	Yes	T/O	0.24s	30.00s	T/O	0.28s	30.00s	19%
nec11_false.c	✘	✘	0.29s	Yes	✘	0.13s	0.08s	✘	0.14s	0.12s	12%
nec20_false.c	✘	✘	0.24s	Yes	✘	0.51s	0.37s	✘	0.52s	0.44s	17%
string_false.c	✘	↯	30.00s		—	—	—	—	—	—	—
sum01_bug02_false.c	✘	✘	0.27s	Yes	✘	1.89s	0.82s	✘	1.95s	1.01s	27%
sum01_bug02_sum01_bug02_base.case_false.c	✘	✘	0.26s	Yes	✘	0.45s	1.94s	✘	0.47s	0.80s	20%
sum01_false.c	✘	✘	0.22s	Yes	✘	0.46s	0.35s	✘	0.47s	0.30s	22%
sum03_false.c	✘	✘	2.45s	Yes	✘	0.65s	0.65s	✘	0.70s	0.80s	32%
sum04_false.c	✘	✘	0.05s	Yes	✘	0.35s	0.19s	✘	0.36s	0.25s	24%
sum_array_false.c	✘	↯	30.00s	Yes	T/O	0.59s	30.00s	T/O	0.64s	30.00s	28%
terminator_01_false.c	✘	✘	0.28s	Yes	✘	0.12s	0.13s	✘	0.12s	0.15s	12%
terminator_02_false.c	✘	✘	3.56s		—	—	—	—	—	—	—
terminator_03_false.c	✘	✘	0.42s		—	—	—	—	—	—	—
trex01_false.c	✘	✘	2.69s		—	—	—	—	—	—	—
trex02_false.c	✘	✘	0.66s		—	—	—	—	—	—	—
trex03_false.c	✘	✘	8.21s	Yes	✘	3.96s	0.70s	✘	3.98s	1.65s	54%
verisec_NetBSD-libc__loop_false.c	✘	✘	10.45s		—	—	—	—	—	—	—
verisec_OpenSER__cases1_stripFullBoth_arr_false.c	✘	T/O	30.00s	Yes	T/O	1.03s	30.00s	T/O	1.20s	30.00s	76%
verisec_sendmail__tTflag_arr_one_loop_false.c	✘	T/O	30.00s		—	—	—	—	—	—	—
vogal_false.c	✘	↯	30.00s	Yes	T/O	1.62s	30.00s	T/O	1.83s	30.00s	68%
while_infinite_loop_4_false.c	✘	✘	3.03s		—	—	—	—	—	—	—
Total	32	20	394.96s	18	11	15.79s	197.94s	12	16.51s	173.74s	

Key: Safe: ✔, Unsafe: ✘, Timeout: T/O, Crash: ↯, Incomplete (unable to prove safety or find a bug): ?

Table 4.3: Detailed experimental results for unsafe SV-COMP benchmarks

		CBMC		Accelerated?	CBMC + Acceleration			CBMC + Acceleration + Trace Automata			
Name	Expected	Result	Time(s)		Result	Acceleration time (s)	Checking time (s)	Result	Acceleration time (s)	Checking time (s)	Size increase
Crafted											
array_safe1	✔	?	0.20s	Yes	?	0.15s	0.27s	✔	0.16s	0.08s	11%
array_safe2	✔	?	0.08s	Yes	T/O	0.14s	30.00s	✔	0.13s	0.09s	10%
array_safe3	✔	?	0.48s	Yes	?	0.12s	0.28s	✔	0.14s	0.07s	15%
array_safe4	✔	?	0.49s	Yes	?	0.12s	0.19s	✔	0.14s	0.05s	13%
const_safe1	✔	?	0.27s	Yes	?	0.15s	0.06s	✔	0.15s	0.08s	12%
diamond_safe1	✔	?	0.51s	Yes	?	0.18s	0.15s	✔	0.18s	0.13s	26%
diamond_safe2	✔	?	7.53s	Yes	?	0.66s	0.77s	✔	0.66s	0.45s	32%
functions_safe1	✔	?	0.06s	Yes	?	0.13s	0.08s	✔	0.15s	0.06s	12%
multivar_safe1	✔	?	0.40s	Yes	?	0.18s	0.08s	✔	0.19s	0.07s	12%
overflow_safe1	✔	?	0.04s	Yes	?	0.13s	0.06s	✔	0.14s	0.07s	13%
phases_safe1	✔	?	0.04s	Yes	?	0.23s	0.09s	✔	0.25s	0.14s	26%
simple_safe1	✔	?	0.04s	Yes	?	0.14s	0.08s	✔	0.15s	0.06s	13%
simple_safe2	✔	?	1.00s	Yes	?	0.13s	0.09s	✔	0.15s	0.10s	13%
simple_safe3	✔	?	0.24s	Yes	?	0.13s	0.07s	✔	0.14s	0.07s	13%
simple_safe4	✔	?	0.04s	Yes	?	0.16s	0.14s	✔	0.18s	0.07s	13%
Total	15	0	11.42s	15	0	2.75s	32.41s	15	2.91s	1.59s	
array_unsafe1	✘	?	0.49s	Yes	✘	0.12s	0.04s	✘	0.13s	0.06s	14%
array_unsafe2	✘	?	0.09s	Yes	✘	0.15s	10.42s	✘	0.15s	0.11s	10%
array_unsafe3	✘	?	0.48s	Yes	✘	0.14s	0.04s	✘	0.14s	0.06s	14%
const_unsafe1	✘	?	0.05s	Yes	✘	0.12s	0.05s	✘	0.13s	0.07s	12%
diamond_unsafe1	✘	?	0.51s	Yes	✘	0.25s	0.10s	✘	0.24s	0.20s	26%
diamond_unsafe2	✘	?	7.03s	Yes	✘	0.86s	0.88s	✘	0.89s	1.41s	33%
functions_unsafe1	✘	?	0.06s	Yes	✘	0.13s	0.07s	✘	0.12s	0.07s	12%
multivar_unsafe1	✘	?	0.05s	Yes	✘	0.20s	0.12s	✘	0.19s	0.08s	11%
overflow_unsafe1	✘	?	0.05s	Yes	✘	0.13s	0.09s	✘	0.16s	0.08s	13%
phases_unsafe1	✘	?	0.06s	Yes	✘	0.23s	0.10s	✘	0.23s	0.13s	26%
simple_unsafe1	✘	?	0.04s	Yes	✘	0.12s	0.06s	✘	0.15s	0.06s	13%
simple_unsafe2	✘	?	0.04s	Yes	✘	0.12s	0.05s	✘	0.13s	0.06s	13%
simple_unsafe3	✘	?	0.04s	Yes	✘	0.13s	0.06s	✘	0.14s	0.07s	12%
simple_unsafe4	✘	?	0.04s	Yes	✘	0.15s	0.16s	✘	0.15s	0.09s	13%
Total	14	0	9.03s	14	14	2.85s	12.24s	14	2.95s	2.55s	

Key: Safe: ✔, Unsafe: ✘, Timeout: T/O, Crash: ↯, Incomplete (unable to prove safety or find a bug): ?

Table 4.4: Detailed experimental results for crafted benchmarks

Part II

Second-Order Logic and Program Synthesis

Chapter 5

Overview and Preliminaries

Collaborators *The bulk of the work presented in this part has been published in [36, 37, 64]. The latter two of these papers were written in conjunction with Cristina David.*

In this part, we will make use of second-order logic to model various program analysis problems. Our discussion will begin in Chapter 6 by defining a fragment of second-order logic and its associated decision problem, which we will call second-order SAT. We will go on to discuss some of the properties of second-order SAT, in particular that its decision problem is NEXPTIME-complete and that it can be solved with a finite-state program synthesiser. Having made this observation, we will describe an algorithm for finite-state program synthesis and observe that this algorithm gives us an NEXPTIME-complete decision procedure. In the following chapters we will illustrate the richness of second-order SAT by showing how to build different program analysers on top of this decision procedure. In Chapter 7 we will show how many program analysis tasks can be encoded as second-order SAT, including: termination an non-termination proofs, safety proofs and finding deep bugs without unrolling. In Chapter 8 we will introduce a theory for reasoning about programs manipulating singly-linked lists, based on a reduction to SAT. The theory is sound, complete, precise and decidable in NP. We will first show how this theory can be used to express complex invariants for list-manipulating programs, then we will show how the theory can be combined with second-order SAT to automatically infer invariants.

5.1 Background and Notation

In Part I, we analysed C programs by finding closed forms for their loops and inserting these closed forms back into the program. For this purpose it was convenient to think of programs as CFAs, which exposed the control flow explicitly and made it easy to mutate the program. By contrast, this part is concerned with inferring invariants and ranking functions for use in Hoare-style proofs. These proofs amount to checking whether certain logical formulae are valid and consequently our presentation from here on will be logical rather than automata-theoretic and so we will consider programs to be transition systems. A transition system is a pair of a set X, called the state space, and a relation $S \subseteq X \times X$, called the transition relation. If $S(x, x')$, we say that x' is a *successor* of x.

We will often be concerned with sets of states and relations on states. For ease of presentation, we will identify a set with its characteristic function, for example we will write:

$$P(x, y) = \{\langle x, y\rangle \mid x \neq y\}$$

as:

$$P(x, y) \triangleq x \neq y$$

As in Part I, we will primarily concern ourselves with the analysis of loops. Without loss of generality, we will consider all loops to have the following structure:

```
assume(I);

while (G) {
  B;
}

assert(A);
```

Our logical encoding of this loop will be:

- $I(x)$ – a predicate representing the states the loop can begin executing in.
- $G(x)$ – a predicate representing the loop guard, which is true of some state iff the loop would execute.
- $B(x, x')$ – a relation representing the body of the loop.
- $A(x)$ – a predicate that must hold for the assertion at the end of the loop to hold. Note that if a loop has an assertion inside the loop body, it can always be rewritten to have the assertion outside.

5.1.1 Logical concepts

A logic is a collection of rules for manipulating strings of symbols. The logic comes equipped with rules for determining whether some string is a *well formed formula* and whether one formula is a consequence of another. If a formula ψ is a consequence of a formula ϕ, we say that ϕ entails ψ, or:

$$\phi \models \psi$$

A theory T is a collection of formulae that are closed under logical consequence – that is, for every pair of formulae ϕ and ψ, if $\phi \in T$ and $\phi \models \psi$ then $\psi \in T$. We say that every $\phi \in T$ is a *theorem* of T. We will often characterise a theory by assuming first-order logic as a background theory, then describing T as the smallest theory that includes both first-order logic and some set of formulae A – the *axioms* of T. We will often write $\phi \in T$ as $T \models \phi$. When it is clear which theory T we are working in, we will often write $A \models \phi$ to mean $T \cup A \models \phi$, i.e. ϕ is a consequence of T along with the extra axioms A.

5.1.2 C$^-$

All of the theories we will discuss in this part will be at least NP-complete. The canonical NP-complete decision problem is propositional SAT, which asks the question "is this propositional formula satisfiable?" It is conventional to use SAT as a background theory for many program analysis tasks, but we will use another NP-complete theory as our background theory, which we will call C$^-$. The members of C$^-$ are safe, loop-free C programs, i.e. loop-free C programs containing assertions such that no execution causes an assertion to fail. This theory is NP-complete almost by definition and is much easier than SAT to encode verification problems in. The decision problem for this theory is "can any of the assertions in this loop-free C program fail?", which we solve using CBMC [29].

C$^-$ programs can use bit-vector integers, floating-point variables, pointers, structs and arrays. They can have branching, if-then-else, assumptions and assertions, but cannot use loops or backwards jumps.

Chapter 6

Second-Order SAT Solving with Program Synthesis

6.1 Introduction

Program synthesis is the mechanised construction of software that provably satisfies a given specification. Synthesis tools promise to relieve the programmer from thinking about *how* the problem is to be solved; instead, the programmer only provides a compact description of *what* is to be achieved. Foundational research in this area has been exceptionally fruitful, beginning with Alonzo Church's work on the *Circuit Synthesis Problem* in the sixties [28]. Algorithmic approaches to the problem have frequently been connected to automated theorem proving [63, 80]. Recent developments include an application of Craig interpolation to synthesis [57].

We will show that program synthesis in general is undecidable and in fact harder than the halting problem for Turing machines. Decidability can be recovered by restricting the class of programs that are synthesised; in particular, we show that if we only consider programs with finite state spaces our problem becomes NEXPTIME-complete. As part of this complexity analysis, we observe a correspondence between program synthesis and existential second-order logic.

Second-order logic allows quantification over sets and functions, as well as ground terms. This expressive power makes it easy to encode many program analysis problems, for example termination [37], safety [47,93], and superoptimisation [22,49]. If we restrict our language to allow only existential second-order quantification and require a finite universe over which ground terms are interpreted, we obtain the second-order SAT problem, which we formally define in Section 6.2. This problem is similar to another NEXPTIME-complete problem: satisfiability of QFBAPA-REL [99], which uses Presburger arithmetic and quantification over relations. In contrast to this work, second-order SAT use exclusively propostional variables, which we make use of for building bit-precise analyses. After showing how many problems, including all those listed above, can be concisely and naturally encoded as second-order SAT, we show how a finite state program synthesiser can be used as a second-order SAT solver.

This complexity result provides a direct bridge between logic and synthesis, allowing existing program synthesisers [49, 95] to be immediately applied to a wide range of problems, e.g. those stemming from existing work on solving second-order logic constraints for program analysis [10, 14, 47]. This connection is relevant as recent work on syntax guided synthesis [2] promises to greatly raise the profile of program synthesis and usher in a generation of new synthesis tools.

Having developed the theory of second-order SAT, we extend existing approaches to program synthesis to build a fully automatic, sound and complete algorithm for synthesising loop-free C programs. This choice of formalism (loop-free C programs) makes it particularly easy to encode second-order SAT constraints arising from program analysis problems. The resulting synthesiser uses a novel combination of bounded model checking, explicit state model checking and genetic programming. We then prove that our algorithm is optimal in the following senses:

- Encoding programs as loop-free C programs is asymptotically optimal in size.
- If a specification is satisfiable, our algorithm produces the smallest correct program.
- If a specification is satisfiable, the runtime of our algorithm is predominantly a function of the size of the program that is synthesised.

6.2 Preliminaries

In this section we will recall some well known decision problems along with their associated complexity classes. We will then define an extension of propositional SAT that we will call second-order SAT. The section concludes with a proof that second-order SAT is NEXPTIME-complete.

Definition 19 (Propositional SAT)**.**

$$\exists x_1 \ldots x_n . \sigma$$

Where the x_i range over Boolean values and σ is a quantifier-free propositional formula whose variables are the x_i.

Checking the truth of an instance of Definition 19 is NP-complete.

Definition 20 (First-Order Propositional SAT or QBF)**.**

$$Q_1 x_1 . Q_2 x_2 \ldots Q_n x_n . \sigma$$

Where the Q_i are either $\exists$ or $\forall$. The x_i and σ are as in Definition 19.

Checking the truth of an instance of Definition 20 is PSPACE-complete.

Now we turn our attention to second-order logic. Second-order logic allows quantification over sets as well as objects.

Definition 21 (Second-Order SAT)**.**

$$\exists S_1 \ldots S_m . Q_1 x_1 \ldots Q_n x_n . \sigma$$

Where the S_i range over predicates. Each S_i has an associated arity $\text{ar}(S_i)$ and $S_i \subseteq \mathbb{B}^{\text{ar}(S_i)}$. The remainder of the formula is an instance of Definition 20, except that the quantifier-free part (σ) may refer to both the first-order variables x_i and the second-order variables S_i.

Example 7. The following is a second-order SAT formula:

$$\exists S . \forall x_1, x_2 . S(x_1, x_2) \rightarrow S(x_2, x_1)$$

This formula is satisfiable and is satisfied by any symmetric relation.

Theorem 22 (Fagin's Theorem [43])**.** *The class of structures A recognisable in time $|A|^k$, for some k, by a nondeterministic Turing machine is exactly the class of structures definable by existential second-order sentences.*

Theorem 23 (Second-Order SAT is NEXPTIME-complete)**.** *For an instance of Definition 21 with n first-order variables, checking the truth of the formula is NEXPTIME-complete.*

Proof. We will apply Theorem 22. To do so we must establish the size of the universe implied by Theorem 22. Since Definition 21 uses n Boolean variables, the universe is the set of interpretations of n Boolean variables. This set has size 2^n, and so by Theorem 22, Definition 21 defines exactly the class sets recognisable in $(2^n)^k$ time by a nondeterministic Turing machine. This is the class NEXPTIME, and so checking validity of an arbitrary instance of Definition 21 is NEXPTIME-complete. $\square$

For an alternative proof, consider a Turing machine M. For a particular run of M we can construct a relation $f(k, q, h, j, t)$ defined such that after k steps M is in state q, with its head at position h and tape cell j containing the symbol t. If M halts within 2^n steps on an input of length n, the values of all the variables in this relation are bounded by 2^n, which means they can be written down using n bits. The details of creating a first-order formula constraining f to reflect the behaviour of M are left to the reader.

6.3 Decidability and Complexity of Program Synthesis

The program synthesis problem can be informally described as follows: given a specification, find a program which satisfies that specification for all inputs. In order to define this problem formally, we need to identify what a specification is, what the program we are synthesising is, and what an input is.

6.3.1 General Program Synthesis

For the general case of program synthesis, we will say that:

- The program we wish to synthesise is a Turing machine computing a total function which takes as input a natural number and produces another natural number.
- A specification is a computable function $\sigma(P, x)$ whose arguments are P – the index of some Turing machine, and x a natural number.

The synthesis problem is then that of finding some program P such that the synthesis formula of Definition 24 is true, where we introduce a function $H(P, x)$ which returns true iff the Turing machine with index P halts on input x.

Definition 24 (Synthesis Formula)**.**

$$\forall x \in \mathbb{N} . \sigma(P, x) \wedge H(P, x)$$

The decision problem associated with program synthesis is to determine whether such a P exists for a given σ. We can see immediately that this decision problem is equivalent to the halting problem for an oracle machine with access to the oracle H. It is a well known result that an oracle machine cannot solve its own halting problem, and so the program synthesis problem is undecidable even if we have access to an oracle solving the halting problem. In other words, the general synthesis problem has Turing degree $0''$, making it strictly harder than the halting problem for Turing machines.

In order to recover decidability, we can restrict our scope and consider a finite version of the synthesis problem in which the programs we wish to synthesise have a finite state space $\mathcal{S}$. Since we require our programs to halt on all inputs, we can identify each program P with the total function $f : \mathcal{S} \to \mathcal{S}$ it computes.

6.3.2 Program Encodings

Now we turn to the problem of how to encode such finite-state programs. For the remainder of this thesis, we will encode finite-state programs as loop-free imperative programs consisting of a sequence of instructions, each instruction consisting of an opcode and a tuple of operands. The opcode specifies which operation is to be performed and the operands are the arguments on which the operation will be performed. We allow an operand to be one of: a constant literal, an input to the program, or the result of some previous instruction. Such a program has a natural correspondence with a combinational circuit.

A sequence of instructions is certainly a natural encoding of a program, but we might wonder if it is the *best* encoding. We can show that for a reasonable set of instruction types (i.e. valid opcodes), this encoding is optimal in a sense we will now discuss. An encoding scheme E takes a function f and assigns it a name s. For a given ensemble of functions F we are interested in the worst-case behaviour of the encoding E, that is we are interested in the quantitiy

$$|E(F)| = \max\{|E(f)| \mid f \in F\}.$$

If for every encoding E', we have that

$$|E(F)| = |E'(F)|$$

then we say that E is an *optimal encoding* for F. Similarly if for every encoding E', we have

$$O(|E(F)|) \subseteq O(|E'(F)|)$$

we say that E is an *asymptotically optimal encoding* for F.

Lemma 25 (Languages with ITE are Universal and Optimal Encodings for Finite Functions)**.** *For an imperative programming language including instructions for testing equality of two values (EQ) and an if-then-else (ITE) instruction, any total function $f : \mathcal{S} \to \mathcal{S}$ can be computed by a program of size $O(|\mathcal{S}| \log |\mathcal{S}|)$ bits.*

Proof. The function f is computed by the following program:

```
t1 = EQ(x, 1)
t2 = ITE(t1, f(1), f(0))
t3 = EQ(x, 2)
t4 = ITE(t3, f(2), t2)
...
```

Each operand can be encoded in $\log_2(|\mathcal{S}| + l) = \log_2(3 \times |\mathcal{S}|)$ bits. So each instruction can be encoded in $O(\log |\mathcal{S}|)$ bits and there are $O(|\mathcal{S}|)$ instructions in the program, so the whole program can be encoded in $O(|\mathcal{S}| \log |\mathcal{S}|)$ bits. □

Lemma 26. *Any representation that is capable of encoding an arbitrary total function $f : \mathcal{S} \to \mathcal{S}$ must require at least $O(|\mathcal{S}| \log |\mathcal{S}|)$ bits to encode some functions.*

Proof. There are $|\mathcal{S}|^{|\mathcal{S}|}$ total functions $f : \mathcal{S} \to \mathcal{S}$. Therefore by the pigeonhole principle, any encoding that can encode an arbitrary function must use at least $\log_2(|\mathcal{S}|^{|\mathcal{S}|}) = O(|\mathcal{S}| \log_2 |\mathcal{S}|)$ bits to encode some function. □

From Lemma 25 and Lemma 26, we can conclude that *any* set of instruction types that include ITE is an asymptotically optimal function encoding for total functions with finite domains.

6.3.3 Finite Program Synthesis

To formally define the finite synthesis problem, we will require that the inputs x are drawn from some finite domain $\mathcal{D}$ and that P and σ are loop-free imperative programs. We will allow σ to make use of a `CALL` instruction with which it can call P as a subroutine. We then define the finite-state synthesis decision problem as checking the truth of Definition 27.

Definition 27 (Finite Synthesis Formula)**.**

$$\exists P. \forall x \in \mathcal{D}. \sigma(x)$$

We will now show that each instance of Definition 21 can be reduced in polynomial time to an instance of Definition 27.

Theorem 28 (Second-Order SAT is Polynomial Time Reducible to Finite Synthesis)**.** *Every instance of Definition 21 is polynomial time reducible to an instance of Definition 27.*

Proof. We first Skolemise the instance of definition 21 to produce an equisatisfiable second-order sentence with the first-order part only having universal quantifiers (i.e. bring the formula into Skolem normal form). This process will have introduced a function symbol for each first order existentially quantified variable and taken linear time. Now we just existentially quantify over the Skolem functions, which again takes linear time and space. The resulting formula is an instance of Definition 27. □

Corollary 29. *Finite-state program synthesis is NEXPTIME-complete.*

6.4 Synthesising Finite-State Programs

In this section we will present a sound and complete algorithm for the finite-state synthesis decision problem, as well as details of our implementation. In the case that a specification is satisfiable, our algorithm produces a minimal satisfying program. We begin by describing a general purpose synthesis procedure (Section 6.4.1), then detail how this general purpose procedure is instantiated for synthesising finite-state programs. For the latter part, we will describe the logic on which our system is built and how programs are encoded in our system (Section 6.4.2). We then describe the algorithm we use to search the space of possible programs (Sections 6.4.3, 6.4.4 and 6.4.5), some optimisations we found to be essential (Section 6.4.6) and conclude with a proof of soundness and complexity bounds (Section 6.4.7).

6.4.1 General Purpose Synthesis Algorithm

We use Counterexample Guided Inductive Synthesis (CEGIS) [22,94,95] to find a program satisfying our specification. The core of the CEGIS algorithm is the refinement loop given in Figure 6.1 and detailed in Algorithm 1.

The algorithm is divided into two procedures: SYNTH (see Figure 6.5) and VERIF, which interact via a finite set of test vectors INPUTS.

The SYNTH procedure tries to find an existential witness P that satisfies the partial specification:

$$\exists P . \forall x \in \text{INPUTS} . \sigma(x, P)$$

Algorithm 1 Abstract refinement algorithm

```
1:  function SYNTH(inputs)
2:      $(i_1, \ldots, i_N) \leftarrow$ inputs
3:      query $\leftarrow \exists P.\sigma(i_1, P) \land \ldots \land \sigma(i_N, P)$
4:      result $\leftarrow$ decide(query)
5:      if result.satisfiable then
6:          return result.model
7:      else
8:          return unsatisfiable

9:  function VERIF(P)
10:     query $\leftarrow \exists x.\neg\sigma(x, P)$
11:     result $\leftarrow$ decide(query)
12:     if result.satisfiable then
13:         return result.model
14:     else
15:         return valid

16: function REFINEMENT LOOP
17:     inputs $\leftarrow \emptyset$
18:     loop
19:         candidate $\leftarrow$ SYNTH(inputs)
20:         if candidate = UNSAT then
21:             return unsatisfiable
22:         res $\leftarrow$ VERIF(candidate)
23:         if res = valid then
24:             return candidate
25:         else
26:             inputs $\leftarrow$ inputs $\cup$ res
```

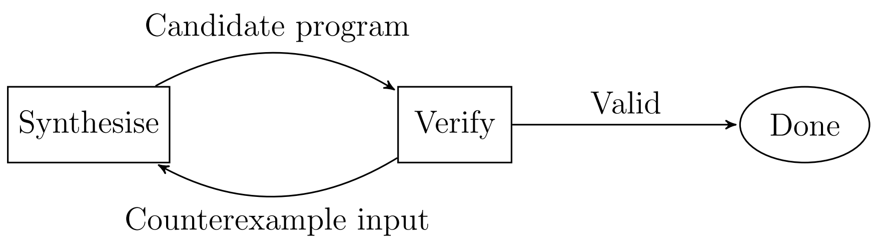


Figure 6.1: Abstract synthesis refinement loop

If SYNTH succeeds in finding a witness P, this witness is a candidate solution to the full synthesis formula. We pass this candidate solution to VERIF which determines whether it does satisfy the specification on all inputs by checking satisfiability of the verification formula:

$$\exists x.\neg\sigma(x, P)$$

If this formula is unsatisfiable, the candidate solution is in fact a solution to the synthesis formula and so the algorithm terminates. Otherwise, the witness x is an input on which the candidate solution fails to meet the specification. This witness x is added to the INPUTS set and the loop iterates again. It is worth noting that each iteration of the loop adds a new input to the set of inputs being used for synthesis. If the full set of inputs X is finite, this means that the refinement loop can only iterate a finite number of times.

6.4.2 Finite-State Synthesis

We will now show how the generic construction of Section 6.4.1 can be instantiated to produce a useful finite-state program synthesiser. A natural choice for such a synthesiser would be to work in the logic of quantifier-free propositional formulae and to use a propositional SAT or SMT-$\mathcal{BV}$ solver as the decision procedure. However we propose a slightly different tack, which is to use a decidable fragment of C as a "high level" logic. We call this fragment C$^-$.

C$^-$

We will now describe the logic we use to express our synthesis formula. The logic is a subset of C that we call C$^-$. The characteristic property of a C$^-$ program is that safety can be decided for it using a single query to a Bounded Model Checker. A C$^-$ program is just a C program with the following syntactic restrictions: all loops in the program must have a constant bound; all recursion in the program must be limited to a constant depth; all arrays must be statically allocated (i.e. not using `malloc`), and be of constant size. Additionally, C$^-$ programs may use nondeterministic values, assumptions and arbitrary-width types.

Since each loop is bounded by a constant, and each recursive function call is limited to a constant depth, a C$^-$ program necessarily terminates and in fact does so in $O(1)$ time. If we call the largest loop bound k, then a Bounded Model Checker with an unrolling bound of k will be a complete decision procedure for the safety of the program. For a C$^-$ program of size l and with largest loop bound k, a Bounded

Model Checker will create a SAT problem of size $O(lk)$. Conversely, a SAT problem of size s can be converted trivially into a loop-free C$^-$ program of size $O(s)$. The safety problem for C$^-$ is therefore NP-complete, which means it can be decided fairly efficiently for many practical instances.

Encoding the Specification in C$^-$

To instantiate the abstract synthesis algorithm in C$^-$ we must express X, Y, σ and P in C$^-$, then ensure that we can express the validity of the synthesis formula as a safety property of the resulting C$^-$ program.

Our encoding for these pieces is the following:

- X is the set of N-tuples of 32-bit bitvectors. This is written in C$^-$ as the type `int[N]`.
- Y is the set of M-tuples of 32-bit bitvectors, which is written in C$^-$ as the type `int[M]`.
- σ is a pure function with type $X \times Y \to \text{Bool}$. The C$^-$ signature of this function is `int check(int in[N], int out[M])`. This function is the only component supplied by the user.
- P is written in a simple RISC-like language $\mathcal{L}$, whose syntax is given in Fig. 6.2. Programs in $\mathcal{L}$ have type $X \to Y$ and are represented in C$^-$ as objects of type `prog_t`, shown in Fig. 6.4.
- We supply an interpreter for $\mathcal{L}$ which is written in C$^-$. The type of this interpreter is $(X \to Y) \times X \to Y$ and the C$^-$ signature is `void exec(prog_t p, int in[N], int out[M])`. Here, `out` is an output parameter.

The exact details of how we encode an $\mathcal{L}$-program are given in Sec. 6.4.2. We must now express the SYNTH and VERIF formulae as safety properties of C$^-$ programs, which is given in Fig. 6.3.

In order to determine the validity of the SYNTH formula, we can check the SYNTH program for safety. The SYNTH program is a C$^-$ program, which means we can check its safety with Bounded Model Checking (BMC) as implemented in the CBMC tool. There are alternative approaches we can use to check the safety of SYNTH.C, each of which boils down to searching for a candidate assignment to p that makes the assertion in SYNTH.C fail.

Integer arithmetic instructions:

```
add a b     sub a b     mul a b     div a b
neg a       mod a b     min a b     max a b
```

Bitwise logical and shift instructions:

```
and  a b    or   a b    xor a b
lshr a b    ashr a b    not a
```

Unsigned and signed comparison instructions:

```
le  a b    lt  a b    sle  a b
slt a b    eq  a b    neq  a b
```

Miscellaneous logical instructions:

```
implies a b    ite a b c
```

Floating-point arithmetic:

```
fadd a b    fsub a b    fmul a b    fdiv a b
```

Figure 6.2: The language $\mathcal{L}$

```
void synth() {
  prog_t p = nondet();
  int in[N], out[M];

  assume(wellformed(p));

  in = test1;
  exec(p, in, out);
  assume(check(in, out));
  ...
  in = testN;
  exec(p, in, out);
  assume(check(in, out));

  assert(false);
}
```

```
void verif(prog_t p) {
  int in[N] = nondet();
  int out[M];

  exec(p, in, out);
  assert(check(in, out));
}
```

Figure 6.3: The SYNTH and VERIF formulae expressed as a C^{-} program.

```
typedef $\mathcal{BV}(4)$ op_t;                       // An opcode
typedef $\mathcal{BV}(w)$ word_t;                     // An $\mathcal{L}$-word
typedef $\mathcal{BV}(\log_2\lceil c+l+a\rceil)$ param_t;  // An operand

struct prog_t {
  op_t ops[$l$];              // The opcodes
  param_t params[$l*2$];   // The operands
  word_t consts[$c$];        // The program constants
}
```

Figure 6.4: The C$^-$ structure we use to encode an $\mathcal{L}$ program

Encoding a Candidate Solution in C$^-$

Solutions to a synthesis specification are $\mathcal{L}$ programs. The exact C$^-$ encoding of an $\mathcal{L}$ program is shown in Fig. 6.4. The `prog_t` structure encodes a program, which is a sequence of instructions. The parameter a is the number of arguments the program takes. The ith instruction has opcode `ops[i]`, left operand `params[i*2]` and right operand `params[i*2 + 1]`. An operand refers to either a program constant, a program argument or the result of a previous instruction, and its value is determined at runtime as follows:

$$val(x) = \begin{cases} x < a & \text{the } x^{\text{th}} \text{ program argument} \\ a \leq x < a+c & \texttt{consts}[x-a] \\ x \geq a+c & \text{the result of the } (x-a-c)^{\text{th}} \text{ instruction} \end{cases}$$

A program is well formed if no operand refers to the result of an instruction that has not been computed yet, and if each opcode is valid. We add a well-formedness constraint of the form `params[i] <= (a+c+2*i)` for each instruction. It should be noted that this requires a linear number of well-formedness constraints. If all of these constraints are satisfied, the program is well-formed in the sense.

6.4.3 Candidate Generation Strategies

The remit of the SYNTH portion of the CEGIS loop, as shown in Figure 6.1, is to generate candidate programs. There are many possible strategies for finding these candidates; we employ the following strategies in parallel:

Explicit Proof Search. The simplest strategy for finding candidates is to just exhaustively enumerate them all, starting with the shortest and progressively increasing

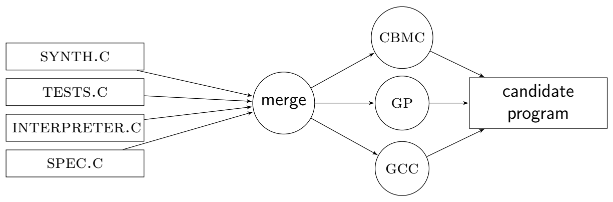


Figure 6.5: Schematic diagram of SYNTH

the number of instructions. This strategy is implemented by the ExplicitSearch routine. Since the set of $\mathcal{L}$-programs is recursively enumerable, this procedure is complete.

Symbolic Bounded Model Checking. Another complete method for generating candidates is to simply use BMC on the SYNTH.C program. As with explicit search, we must progressively increase the length of the $\mathcal{L}$-program we search for in order to get a complete search procedure.

Genetic Programming and Incremental Evolution. Our final strategy is genetic programming (GP) [24, 73]. GP provides an adaptive way of searching through the space of $\mathcal{L}$-programs for an individual that is "fit" in some sense. We measure the fitness of an individual by counting the number of tests in INPUTS for which it satisfies the specification.

To bootstrap GP in the first iteration of the CEGIS loop, we generate a population of random $\mathcal{L}$-programs. We then iteratively evolve this population by applying the genetic operators CROSSOVER and MUTATE. Crossover combines selected existing programs into new programs, whereas MUTATE randomly changes parts of a single program. Fitter programs are more likely to be selected.

Rather than generating a random population at the beginning of each subsequent iteration of the CEGIS loop, we start with the population we had at the end of the previous iteration. The intuition here is that this population contained many individuals that performed well on the k inputs we had before, so they will probably continue to perform well on the $k + 1$ inputs we have now. In the parlance of evolutionary programming, this is known as incremental evolution [46].

6.4.4 Parameterising the Program Space

In order to search the space of candidate programs, we parametrise the language $\mathcal{L}$, inducing a lattice of progressively more expressive languages. We start by attempting to synthesise a program at the lowest point on this lattice and increase the parameters of $\mathcal{L}$ until we reach a point at which the synthesis succeeds.

As well as giving us an automatic search procedure, this parametrisation greatly increases the efficiency of our system since languages low down the lattice are very easy to decide safety for. If a program can be synthesised in a low-complexity language, the whole procedure finishes much faster than if synthesis had been attempted in a high-complexity language.

Program Length: l The first parameter we introduce is program length, denoted by l. At each iteration we synthesise programs of length exactly l. We start with $l = 1$ and increment l whenever we determine that no program of length l can satisfy the specification. When we do successfully synthesise a program, we are *guaranteed that it is of minimal length* since we have previously established that no shorter program is correct.

Word Width: w An $\mathcal{L}$-program runs on a virtual machine (the $\mathcal{L}$-machine) that has its own set of parameters. The only relevant parameter is the *word width* of the $\mathcal{L}$-machine, that is, the number of bits in each internal register and immediate constant. This parameter is denoted by w. The size of the final SAT problem generated by CBMC scales polynomially with w, since each intermediate C variable corresponds to w propositional variables.

It is often the case that a program which satisfies the specification on an $\mathcal{L}$-machine with $w = k$ will continue to satisfy the specification when run on a machine with $w > k$. For example, the program in Fig. 6.6 isolates the least-significant bit of a word. This is true irrespective of the word size of the machine it is run on – it will isolate the least-significant bit of an 8-bit word just as well as it will a 32-bit word. An often successful strategy is to synthesise a program for an $\mathcal{L}$-machine with a small word size and then to check whether the same program is correct when run on an $\mathcal{L}$-machine with a full-sized word.

The only wrinkle here is that we will sometimes synthesise a program containing constants. If we have synthesised a program with $w = k$, the constants in the program will be k-bits wide. To extend the program to an n-bit machine (with $n > k$), we need some way of deriving n-bit-wide numbers from k-bit ones. We have several strategies for this and just try each in turn. Our strategies are shown in Fig. 6.7. $\mathcal{BV}(v, n)$ denotes an n-bit wide bitvector holding the value v and $b \cdot c$ means the concatenation of bitvectors b and c.

```
int isolate_lsb(int x) {
  return x & -x;
}
```

Example:

x	=	1	0	1	1	1	0	1	0
-x	=	0	1	0	0	0	1	1	0
x & -x	=	0	0	0	0	0	0	1	0

Figure 6.6: A tricky bitvector program

$$\begin{array}{rclrcl}
\mathcal{BV}(m,m) & \rightarrow & \mathcal{BV}(n,n) & \mathcal{BV}(x,m) & \rightarrow & \mathcal{BV}(x,n) \\
\mathcal{BV}(m-1,m) & \rightarrow & \mathcal{BV}(n-1,n) & \mathcal{BV}(x,m) & \rightarrow & \mathcal{BV}(x,m)\cdot\mathcal{BV}(0,n-m) \\
\mathcal{BV}(m+1,m) & \rightarrow & \mathcal{BV}(n+1,n) & \mathcal{BV}(x,m) & \rightarrow & \underbrace{\mathcal{BV}(x,m)\cdot\ldots\cdot\mathcal{BV}(x,m)}_{\frac{n}{m}\text{times}}
\end{array}$$

Figure 6.7: Rules for extending an m-bit wide number to an n-bit wide one.

Sometimes a program will be correct for some particular word width w, but is not correct for $w' > w$ even if the constants are replaced with appropriate ones. When we detect this situation, we increase w and continue synthesising.

Number of Constants: c Instructions in $\mathcal{L}$ take either one or two operands. Since any instruction whose operands are all constants can always be eliminated (since its result is a constant), we know that a loop-free program of minimal length will not contain any instructions with two constant operands. Therefore the number of constants that can appear in a minimal program of length l is at most l. By minimising the number of constants appearing in a program, we are able to use a particularly efficient program encoding that speeds up the synthesis procedure substantially. The number of constants used in a program is the parameter c.

$\mathcal{L}$ is an SSA, three-address instruction set[1]. Destination registers are implicit and a fresh register exists for each instruction to write its output to. A naïve way to encode $\mathcal{L}$ instructions is to have an opcode and two operands, where each operand is either a register (i.e., a program argument or the result of a previous instruction), or an immediate constant.

In this encoding, each opcode requires $\lceil \log_2 I \rceil$ bits to encode, where I is the number of instruction types in $\mathcal{L}$. Each operand can be encoded using $\log_2 w$ bits, where w is the $\mathcal{L}$-machine word width, plus one bit to specify whether the operand is a register name or an immediate constant. One instruction can therefore be encoded

[1] We experimented with implementing $\mathcal{L}$ as a stack machine, expecting the programs to be smaller and synthesis to be faster as a result. We saw the opposite effect – the more complex interpreter led to much slower synthesis.

using $\lceil \log_2 I \rceil + 2w + 2$ bits. For an n-instruction program, we need

$$\lceil n \log_2 I \rceil + 2nw + 2n$$

bits to encode the entire program.

If we instead limit the number of constants that can appear in the program, our operands can be encoded using fewer bits. For an n-instruction program using c constants and taking a arguments as inputs, each operand can refer to a program argument, the result of a previous instruction or a constant. This can be encoded using $\lceil \log_2(c + a + n - 1) \rceil$ bits, which means each instruction can be encoded in $\lceil \log_2 I \rceil + \lceil \log_2(c + a + n - 1) \rceil$ and the full program needs

$$\lceil n \log_2 I \rceil + \lceil n \log_2(c + a + n - 1) \rceil + cw$$

bits to encode.

We give an example. Our language $\mathcal{L}$ has 15 instruction types, so each opcode is 4 bits. For a 10-instruction program over 1 argument, using 2 constants on a 32-bit word machine the first encoding requires $10 * (4 + 32 + 1 + 32 + 1) = 700$ bits. Using the second encoding, each operand can be represented using $\log_2(2 + 1 + 10 - 1) = 4$ bits, and the entire program requires 184 bits. This is a substantial reduction in size and when the desired program requires only few constants this can lead to a very significant speed up.

As with program length, we progressively increase the number of constants in our program. We start by trying to synthesise a program with no constants, then if that fails we attempt to synthesise using one constant and so on until we reach $c = l$.

6.4.5 Searching the Program Space

The key to our automation approach is to come up with a sensible way in which to adjust the $\mathcal{L}$-parameters in order to cover all possible programs. After each round of SYNTH, we may need to adjust the parameters. The logic for these adjustments is shown as a tree in Fig. 6.8.

Whenever SYNTH fails, we consider which parameter might have caused the failure. There are two possibilities: either the program length l was too small, or the number of allowed constants c was. If $c < l$, we just increment c and try another round of synthesis, but allowing ourselves an extra program constant. If $c = l$, there is no point in increasing c any further. This is because no minimal $\mathcal{L}$-program has $c > l$, for if it did there would have to be at least one instruction with two constant operands.

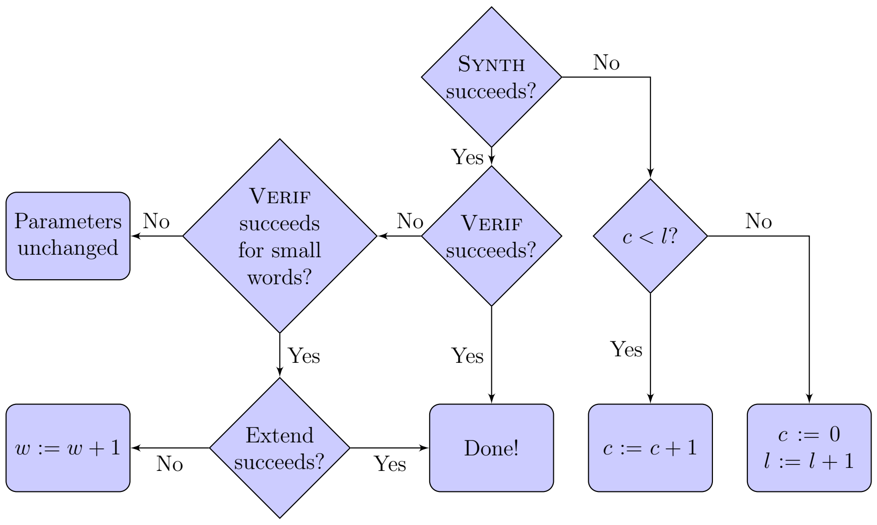


Figure 6.8: Decision tree for increasing parameters of $\mathcal{L}$.

This instruction could be removed (at the expense of adding its result as a constant), contradicting the assumed minimality of the program. So if $c = l$, we set c to 0 and increment l, before attempting synthesis again.

If SYNTH succeeds but VERIF fails, we have a candidate program that is correct for some inputs but incorrect on at least one input. However, it may be the case that the candidate program is correct for *all* inputs when run on an $\mathcal{L}$-machine with a small word size. For example, we may have synthesised a program which is correct for all 8-bit inputs, but incorrect for some 32-bit input. If this is the case (which we can determine by running the candidate program through VERIF using the smaller word size), we may be able to produce a correct program for the full $\mathcal{L}$-machine by using the constant extension rules shown in Fig. 6.7. If constant generalization is able to find a correct program, we are done. Otherwise, we need to increase the word width of the $\mathcal{L}$-machine we are currently synthesising for.

6.4.6 Optimisations

Two optimisations we have found to be very important to the performance of our synthesiser are the following:

Cache binaries We ensure that we do not run GCC more times than necessary, since we have observed compilation time to be relatively expensive. This means that for the phases using native code (explicit-state model checking and the stochastic methods), we compiled the specification once and then just execute the resulting binary in each iteration of the SYNTH and VERIF phases.

Emit C code when possible In the VERIF stage, we can emit the struct-based representation of an $\mathcal{L}$-program along with the code for the interpreter and check the resulting program. Alternatively, since an $\mathcal{L}$-program can be trivially translated to a straight-line C program, we can emit the program as C instead. This results in a much smaller program that is more amenable to optimisation by the compiler and CBMC.

6.4.7 Soundness and Complexity

We will now show that our synthesis algorithm is sound and semi-complete, then will go on to discuss its complexity in terms of the size of the computed solution.

Theorem 30. *Algorithm 1 is sound – if it terminates with witness P, then $P \models \sigma$.*

Proof. The procedure SYNTH terminates only if SYNTH returns "valid". In that case, $\exists x . \neg \sigma(P, x)$ is unsatisfiable and so $\forall x . \sigma(P, x)$ holds. □

Theorem 31. *If the existential first-order theory used to express the specification σ is decidable and the domain of inputs X is finite, Algorithm 1 is semi-complete – if a solution $P \models \sigma$ exists then Algorithm 1 will terminate. However, if no program satisfies the specification, the algorithm may not terminate.*

Proof. If the domain X is finite then the loop in procedure SYNTH can only iterate $|X|$ times, since by this time all of the elements of X would have been added to the inputs set. Therefore if the SYNTH procedure always terminates, Algorithm 1 does as well.

Since the EXPLICITSEARCH routine enumerates all programs (as can be seen by induction on the program length l), it will eventually enumerate a program that meets the specification on whatever set of inputs are currently being tracked, since by assumption such a program exists. Since the first-order theory is decidable, the query in VERIF will succeed for this program, causing the algorithm to terminate. The set of correct programs is therefore recursively enumerable and Algorithm 1 enumerates this set, so it is semi-complete. □

Corollary 32. *Since safety of C^{-} programs is decidable, Algorithm 1 is semi-complete when instantiated with C^{-} as a background theory.*

We will now show that the number of iterations of the CEGIS loop is a function of the Kolmogorov complexity of the synthesised program. We argue that this gives our procedure various desirable qualities in practical applications. We first recall the definition of the Kolmogorov complexity of a function f:

Definition 33 (Kolmogorov complexity)**.** The Kolmogorov complexity $K(f)$ is the length of the shortest program that computes f.

We can extend this definition slightly to talk about the Kolmogorov complexity of a synthesis problem in terms of its specification:

Definition 34 (Kolmogorov complexity of a synthesis problem)**.** The Kolmogorov complexity of a program specification $K(\sigma)$ is the length of the shortest program P such that $P \models \sigma$.

Let us consider the number of iterations of the CEGIS loop n required for a specification σ. Since we enumerate candidate programs in order of length, we are always synthesising programs with length no greater than $K(\sigma)$ (since when we enumerate the first correct program, we will terminate). So the space of solutions we search over is the space of functions computed by $\mathcal{L}$-programs of length no greater than $K(\sigma)$. Let's denote this set $\mathcal{L}(K(\sigma))$. Since there are $O(2^{K(\sigma)})$ *programs* of length $K(\sigma)$ and some functions will be computed by more than one program, we have $|\mathcal{L}(K(\sigma))| \leq O(2^{K(\sigma)})$.

Each iteration of the CEGIS loop distinguishes at least one incorrect function from the set of correct functions, so the loop will iterate no more than $|\mathcal{L}(K(\sigma))|$ times. Therefore another bound on our runtime is:

$$NTIME\left(2^{K(\sigma)}\right)$$

6.5 Experiments

6.5.1 Experimental Setup

We implemented our fully automatic synthesis procedure in the Kalashnikov tool. The specification language of Kalashnikov is C^{-}, which is rich enough to encode arbitrary second-order SAT formulae. To evaluate the viability of second-order SAT, we used Kalashnikov to solve formulae generated from a variety of problems. Our benchmarks come from superoptimisation, code deobfuscation, floating point verification, ranking function and recurrent set synthesis, and QBF solving. The superoptimisation and code deobfuscation benchmarks were taken from the experiments of [49]; the termination benchmarks were taken from SVCOMP'15 [96] and they include the experiments of Chapter 7; the QBF instances consist of some simple instances created by us and some harder instances taken from [45].

We ran our experiments on a 4-core, 3.30 GHz Core i5 with 8 GB of RAM. For our backend solvers, we used CBMC [66] at SVN revision r3545, with Glucose 3.0 [3] as the SAT solver. Each benchmark was run with a timeout of 180 s. For each category of benchmarks, we report the total number of benchmarks in that category, the number we were able to solve within the time limit, the average specification size (in lines of code), the average solution size (in instructions), the average number of iterations of the CEGIS loop, the average time and total time taken. The results are shown in Table 6.1. It should be understood that in contrast to less expressive logics that might be invoked several times in the analysis of some problem, each of these benchmarks is a "complete" problem from the given problem domain. For example, each of the benchmarks in the termination category requires KALASHNIKOV to prove that a full program terminates, i.e. it must find a ranking function and supporting invariants, then prove that these constitute a valid termination proof for the program being analysed.

The timings show that for the instances where we can find a satisfying assignment, we tend to do so quite quickly (on the order of a few seconds). Furthermore the programs we synthesise are often short, even when the problem domain is very complex, such as for termination or QBF.

Not all of these benchmarks are satisfiable, and in particular around half of the termination benchmarks correspond to attempted proofs that non-terminating programs terminate and vice versa. This illustrates one of the current shortcomings of second-order SAT as a decision procedure: we can only conclude that a formula is unsatisfiable once we have examined candidate solutions up to a very high length bound. Being able to detect unsatisfiability of a second-order SAT formula earlier than this would be extremely valuable. We note that for some formulae we can simultaneously search for a proof of satisfiability and of unsatisfiability. For example, since QBF is closed under negation, we can take a QBF formula ϕ then encode both ϕ and $\neg\phi$ as second-order SAT formulae which we then solve.

To help understand the role of the different solvers involved in the synthesis process, we provide a breakdown of how often each solver "won", i.e. was the first to return an answer. This breakdown is shown in Table 6.2. We see that GP and explicit account for the great majority of the responses, with the load spread fairly evenly between them. This distribution illustrates the different strengths of each solver: GP is very good at generating candidates, explicit is very good at finding counterexamples

Category	#Benchmarks	#Solved	Spec. size	Solution size	Iterations	Avg. time (s)	Total time (s)
Superoptimisation	29	22	19.0	4.1	2.7	7.9	166.1
Termination	78	33	93.5	5.7	14.4	11.8	390.4
QBF (simple)	4	4	12.2	9	1.0	1.8	7.1
QBF (hard)	7	1	5889.0	11.0	2.0	1.5	1.5
Total	113	59	49116	295	536	—	565.2

Table 6.1: Experimental results.

CBMC	Explicit	GP	Total
140 12%	510 46%	504 42%	1183 100%

Table 6.2: How often each solver “wins”.

and CBMC is very good at proving that candidates are correct. The GP and explicit numbers are similar because they are approximately “number of candidates found” and “number of candidates refuted” respectively. The CBMC column is approximately “number of candidates proved correct”. The spread of winners here shows that each of the search strategies is contributing something to the overall search and that the strategies are able to co-operate with each other.

To help understand where the time is spent in our solver, Table 6.3 shows how much time is spent in SYNTH, VERIF and constant generalization. Note that generalization counts towards VERIF’s time. We can see that synthesising candidates takes longer than verifying them, but the ratio of around 2:1 is interesting in that neither phase completely dominates the other in terms of runtime cost. This suggests there is great potential in optimising either of these phases.

SYNTH	VERIF	GENERALIZE	Total
389.2 s 69%	175.8 s 31%	25.6 s 5%	565.2 s 100%

Table 6.3: Where the time is spent.

Chapter 7

Second-Order Liveness and Safety

7.1 Introduction

Logical proofs of a program's properties boil down to generating verification conditions (VCs) from a specification and then checking the validity of the VCs. For a loop-free program, these VCs can be generated from a Floyd-Hoare style proof, by using Dijkstra's weakest pre-conditions or by using symbolic execution to compute a strongest post-condition. If a program contains loops, we must annotate the program to show that certain properties are invariants of the program. In this chapter, we will consider the problem of finding and checking annotations that prove the following:

- Safety – none of the assertions in the program can fail.
- Danger – at least one of the assertions can fail.
- Termination – all of the loops terminate on all inputs.
- Non-termination – some loop does not terminate on some input.

We will generate proofs for each of these properties by encoding the problems as second-order SAT. Section 7.6 will introduce ranking functions, which we will use to prove that loops terminate, and recurrence sets which prove non-termination. This section will also show how termination proofs for individual loops can be composed into a proof for the program as a whole, which includes the case of nested loops. In Section 7.7 we will show how safety invariants can be used to prove safety, then we will define the dual notion of a danger invariant, which constitutes a proof that an error can occur. In contrast to most existing approaches, a danger invariant is a concrete witness to the existence of an error that does not require unrolling the program. In keeping with the theme of this part, all of these objects will be encoded as second-order SAT formulae.

The final section of this chapter, Section 7.8, will show how we can profitable construct and check formulae that we know a priori to be true. In the case of termination, we will take the disjunction of the second-order formula encoding termination and the formula encoding non-termination. We know that exactly one of these formulae will be satisfiable and so the disjunction is certainly satisfiable. However we will show that the solution to this true formula will include sufficient information for us to soundly identify whether the program terminates or not. This technique applies generally to any situation in which we second-order SAT gives us a complete encoding of a particular program property.

As mentioned in Chapter 5, we will be considering loops of the form:

```
assume(I);

while (G) {
  B;
}

assert(A);
```

We will encode this loop with predicates for the initial states: $I(x)$, guard: $G(x)$, body: $B(x, x')$ and assertion: $A(x)$.

7.2 Termination

The halting problem has been of central interest to computer scientists since it was first considered by Turing in 1936 [97]. Informally, the halting problem is concerned with answering the question "does this program run forever, or will it eventually terminate?"

Proving program termination is typically done by finding a *ranking function* for the program states, i.e. a monotone map from the program's state space to a well-ordered set. Historically, the search for ranking functions has been constrained in various syntactic ways, leading to incompleteness, and is performed over abstractions that do not soundly capture the behaviour of physical computers. In this chapter, we present a sound and complete method for deciding whether a program with a fixed amount of storage terminates. Since such programs are necessarily finite state, our problem is much easier than Turing's, but is a better fit for analysing computer programs.

When surveying the area of program termination chronologically, we observe an initial focus on monolithic approaches based on a single measure shown to decrease over all program paths [19, 86], followed by more recent techniques that use termination arguments based on Ramsey's theorem [30, 32, 87]. The latter proof style builds an argument that a transition relation is disjunctively well founded by composing several small well-foundedness arguments. The main benefit of this approach is the simplicity of local termination measures in contrast to global ones. For instance, there are cases in which linear arithmetic suffices when using local measures, while corresponding global measures require nonlinear functions or lexicographic orders.

One drawback of the Ramsey-based approach is that the validity of the termination argument relies on checking the *transitive closure* of the program, rather than a single step. As such, there is experimental evidence that most of the effort is spent in reachability analysis [32, 68], requiring the support of powerful safety checkers: there is a tradeoff between the complexity of the termination arguments and that of checking their validity.

As Ramsey-based approaches are limited by the state of the art in safety checking, recent research shifts back to more complex termination arguments that are easier to check [33, 68]. Following the same trend, we investigate its extreme: *unrestricted* termination arguments. This means that our ranking functions may involve nonlinearity and lexicographic orders: we do not commit to any particular syntactic form, and do not use templates. Furthermore, our approach allows us to *simultaneously* search for proofs of *non-termination*, which take the form of recurrence sets.

Figure 7.1 summarises the related work with respect to the restrictions they impose on the transition relations as well as the form of the ranking functions computed. While it supports the observation that the majority of existing termination analyses are designed for linear programs and linear ranking functions, it also highlights another simplifying assumption made by most state-of-the-art termination provers: that bit-vector semantics and integer semantics give rise to the same termination behaviour. Thus, most existing techniques treat fixed-width machine integers (bit-vectors) and IEEE floats as mathematical integers and reals, respectively [8, 21, 32, 54, 68, 86].

By assuming bit-vector semantics to be identical to integer semantics, these techniques ignore the wrap-around behaviour caused by overflows, which can be unsound. In Section 7.3, we show that integers and bit-vectors exhibit incomparable behaviours

		Program							
		Rationals/Integers		Reals		Bit-vectors		Floats	
		L	NL	L	NL	L	NL	L	NL
Ranking	Linear lexicographic	[8, 19, 33, 86]	-	[76]	-	✓	✓	✓	✓
	Linear non-lexicographic	[21, 32, 54, 68, 75]	[21]	[76]	-	✓ [31]	✓ [31]	✓	✓
	Nonlinear lexicographic	-	-	-	-	✓	✓	✓	✓
	Nonlinear non-lexicographic	[21]	[21]	-	-	✓	✓	✓	✓

Legend: ✓= we can handle; - = no available works; L = linear; NL = nonlinear.

Figure 7.1: Summary of related termination analyses

with respect to termination, i.e. programs that terminate for integers need *not* terminate for bit-vectors and vice versa. Thus, abstracting bit-vectors with integers may give rise to *unsound* and *incomplete* analyses.

7.3 Termination Examples

Figure 7.1 illustrates the most common simplifying assumptions made by existing termination analyses:

(i) programs use only linear arithmetic.

(ii) terminating programs have termination arguments expressible in linear arithmetic.

(iii) the semantics of bit-vectors and mathematical integers are equivalent.

(iv) the semantics of IEEE floating-point numbers and mathematical reals are equivalent.

To show how these assumptions are violated by even simple programs, we draw the reader's attention to the programs in Figure 7.2 and their curious properties:

- Program (a) breaks assumption (i) as it makes use of the bit-wise & operator. Our technique finds that an admissible ranking function is the linear function $R(x) = x$, whose value decreases with every iteration, but cannot decrease indefinitely as it is bounded from below. This example also illustrates the lack of a direct correlation between the linearity of a program and that of its termination arguments.

- Program (b) breaks assumption (ii), in that it has no linear ranking function. We prove that this loop terminates by finding the nonlinear ranking function $R(x) = |x|$.

- Program (c) breaks assumption (iii). This loop is terminating for bit-vectors since x will eventually overflow and become negative. Conversely, the same program is non-terminating using integer arithmetic since $x > 0 \to x + 1 > 0$ for any integer x.

- Program (d) also breaks assumption (iii), but "the other way": it terminates for integers but not for bit-vectors. If each of the variables is stored in an unsigned k-bit word, the following entry state will lead to an infinite loop:

$$M = 2^k - 1, \quad N = 2^k - 1, \quad i = M, \quad j = N - 1$$

- Program (e) breaks assumption (iv): it terminates for reals but not for floats. If x is sufficiently large, rounding error will cause the subtraction to have no effect.

- Program (f) breaks assumption (iv) "the other way": it terminates for floats but not for reals. Eventually x will become sufficiently small that the nearest representable number is 0.0, at which point it will be rounded to 0.0 and the loop will terminate.

Up until this point, we considered examples that are not soundly treated by existing techniques as they don't fit in the range of programs addressed by these techniques. Next, we look at some programs that are handled by existing termination tools via dedicated analyses. We show that our method handles them uniformly, without the need for any special treatment.

- Program (g) is a linear program that is shown in [33] not to admit (without prior manipulation) a lexicographic linear ranking function. With our technique we can find the nonlinear ranking function $R(x) = |x|$.

- Program (h) illustrates conditional termination. When proving program termination we are simultaneously solving two problems: the search for a termination argument, and the search for a supporting invariant [25]. For this loop, we find the ranking function $R(x) = x$ together with the supporting invariant $y = 1$.

- In the terminology of [76], program (i) admits a *multiphase* ranking function, computed from a multiphase ranking template. Multiphase ranking templates are targeted at programs that go through a finite number of phases in their execution. Each phase is ranked with an affine-linear function and the phase is considered to be completed once this function becomes non-positive.

In our setting this type of programs does not need special treatment, as we can find a nonlinear lexicographic ranking function $R(x, y, z) = (x < y, z)$.[1]

As with all of the termination proofs presented in this chapter, the ranking functions above were all found completely automatically.

```
while (x > 0) {
  x = (x - 1) & x;
}
```

(a) Taken from [31].

```
while (x != 0) {
  x = -x / 2;
}
```

(b)

```
while(x > 0) {
  x++;
}
```

(c)

```
while (i<M || j<N) {
  i = i + 1;
  j = j + 1;
}
```

(d) Taken from [84]

```
float x;

while (x > 0.0) {
  x -= 1.0;
}
```

(e)

```
float x;

while (x > 0.0) {
  x *= 0.5;
}
```

(f)

```
while (x != 0) {
  if (x > 0)
    x--;
  else
    x++;
}
```

(g) Taken from [33]

```
y = 1;

while (x > 0) {
  x = x - y;
}
```

(h)

```
while (x>0 && y>0 && z>0){
  if (y > x) {
    y = z;
    x = nondet();
    z = x - 1;
  } else {
    z = z - 1;
    x = nondet();
    y = x - 1;
  }
}
```

(i) Taken from [7]

Figure 7.2: Motivational examples, mostly taken from the literature.

7.4 Termination Proofs

Given a program, we first formalise its termination argument as a ranking function (Section 7.4.1). Subsequently, we discuss bit-vector semantics and illustrate differences between machine arithmetic and integer arithmetic that show that the abstraction of bit-vectors to mathematical integers is unsound (Section 7.5).

[1]This termination argument is somewhat subtle. The Boolean values *false* and *true* are interpreted as 0 and 1, respectively. The Boolean $x < y$ thus eventually decreases, that is to say once a state with $x \geq y$ is reached, x never again becomes greater than y. This means that as soon as the "else" branch of the if statement is taken, it will continue to be taken in each subsequent iteration of the loop. Meanwhile, if $x < y$ has not decreased (i.e., we have stayed in the same branch of the "if"), then z does decrease. Since a Boolean only has two possible values, it cannot decrease indefinitely. Since $z > 0$ is a conjunct of the loop guard, z cannot decrease indefinitely, and so R proves that the loop is well founded.

7.4.1 Termination and Ranking Functions

A program P is represented as a transition system with state space X and transition relation $T \subseteq X \times X$. For a state $x \in X$ with $T(x, x')$ we say x' is a successor of x under T.

Definition 35 (Unconditional termination)**.** A program is said to be *unconditionally terminating* if there is no infinite sequence of states $x_1, x_2, \ldots \in X$ with $\forall i.\ T(x_i, x_{i+1})$.

We can prove that the program is unconditionally terminating by finding a ranking function for its transition relation.

Definition 36 (Ranking function)**.** A function $R : X \to Y$ is a *ranking function* for the transition relation T if Y is a well-founded set with order $>$ and R is injective and monotonically decreasing with respect to T. That is to say:

$$\forall x, x' \in X.T(x, x') \Rightarrow R(x) > R(x')$$

Definition 37 (Linear function)**.** A *linear function* $f : X \to Y$ with $\dim(X) = n$ and $\dim(Y) = m$ is of the form:

$$f(\vec{x}) = M\vec{x}$$

where M is an $n \times m$ matrix.

In the case that $\dim(Y) = 1$, this reduces to the inner product

$$f(\vec{x}) = \vec{\lambda} \cdot \vec{x} + c\ .$$

Definition 38 (Lexicographic ranking function)**.** For $Y = Z^m$, we say that a ranking function $R : X \to Y$ is *lexicographic* if it maps each state in X to a tuple of values such that the loop transition leads to a decrease with respect to the lexicographic ordering for this tuple. The total order imposed on Y is the lexicographic ordering induced on tuples of Z's. So for $y = (z_1, \ldots, z_m)$ and $y' = (z'_1, \ldots, z'_m)$:

$$y > y' \iff \exists i \leq m.z_i > z'_i \wedge \forall j < i.z_j = z'_j$$

We note that some termination arguments require lexicographic ranking functions, or alternatively, ranking functions whose co-domain is a countable ordinal, rather than just $\mathbb{N}$.

7.5 Machine Arithmetic Vs. Peano Arithmetic

Physical computers have bounded storage, which means they are unable to perform calculations on mathematical integers. They do their arithmetic over fixed-width binary words, otherwise known as bit-vectors. For the remainder of this section, we will say that the bit-vectors we are working with are k-bits wide, which means that each word can hold one of 2^k bit patterns. Typical values for k are 32 and 64.

Machine words can be interpreted as "signed" or "unsigned" values. Signed values can be negative, while unsigned values cannot. The encoding for signed values is two's complement, where the most significant bit b_{k-1} of the word is a "sign" bit, whose weight is $-(2^k - 1)$ rather than $2^k - 1$. Two's complement representation has the property that $\forall x. - x = (\sim x) + 1$, where $\sim(\bullet)$ is bitwise negation. Two's complement also has the property that addition, multiplication and subtraction are defined identically for unsigned and signed numbers.

Bit-vector arithmetic is performed modulo 2^k, which is the source of many of the differences between machine arithmetic and Peano arithmetic[2]. To give an example, $(2^k - 1) + 1 \equiv 0 \pmod{2^k}$ provides a counterexample to the statement $\forall x. x + 1 > x$, which is a theorem of Peano arithmetic but not of modular arithmetic. When an arithmetic operation has a result greater than 2^k, it is said to "overflow". If an operation does not overflow, its machine-arithmetic result is the same as the result of the same operation performed on integers.

The final source of disagreement between integer arithmetic and bit-vector arithmetic stems from width conversions. Many programming languages allow numeric variables of different types, which can be represented using words of different widths. In C, a `short` might occupy 16 bits, while an `int` might occupy 32 bits. When a k-bit variable is assigned to a j-bit variable with $j < k$, the result is truncated mod 2^j. For example, if x is a 32-bit variable and y is a 16-bit variable, y will hold the value 0 after the following code is executed:

```
x = 65536;
y = x;
```

As well as machine arithmetic differing from Peano arithmetic on the operators they have in common, computers have several "bitwise" operations that are not taken as primitive in the theory of integers. These operations include the Boolean operators `and, or, not, xor` applied to each element of the bit-vector. Computer programs

[2]ISO C requires that unsigned arithmetic is performed modulo 2^k, whereas the overflow case is undefined for signed arithmetic. In practice, the undefined behaviour is implemented just as if the arithmetic had been unsigned.

often make use of these operators, which are nonlinear when interpreted in the standard model of Peano arithmetic[3].

7.6 Termination as Second-Order Satisfaction

The problem of program verification can be reduced to the problem of finding solutions to a second-order constraint [47, 50]. Our intention is to apply this approach to termination analysis. In this section we show how several variations of both the termination and the non-termination problem can be uniformly defined using second-order SAT.

7.6.1 An Isolated, Simple Loop

We will begin our discussion by showing how to encode in second-order SAT the (non-)termination of a program consisting of a single loop with no nesting. For the time being, a loop $L(G, T)$ is defined by its guard G and body T such that states x satisfying the loop's guard are given by the predicate $G(x)$. The body of the loop is encoded as the transition relation $T(x, x')$, meaning that state x' is reachable from state x via a single iteration of the loop body. For example, the loop in Figure 7.2a is encoded as:

$$\begin{aligned} G(x) &= \{x \mid x > 0\} \\ T(x, x') &= \{\langle x, x' \rangle \mid x' = (x - 1) \,\&\, x\} \end{aligned}$$

We will abbreviate this with the notation:

$$\begin{aligned} G(x) &\triangleq x > 0 \\ T(x, x') &\triangleq x' = (x - 1) \,\&\, x \end{aligned}$$

Unconditional termination. We say that a loop $L(G, T)$ is unconditionally terminating iff it eventually terminates regardless of the state it starts in. To prove unconditional termination, it suffices to find a ranking function for $T \cap (G \times X)$, i.e. T restricted to states satisfying the loop's guard.

Theorem 43. *The loop $L(G, T)$ terminates from every start state iff formula* **[UT]** *(Definition 39, Figure 7.3) is satisfiable.*

[3]Some of these operators can be seen as linear in a different algebraic structure, e.g. `xor` corresponds to addition in the Galois field GF(2^k).

Definition 39 (Unconditional Termination Formula [**UT**]).

$$\exists R.\forall x, x'.G(x) \wedge T(x,x') \rightarrow R(x) > 0 \wedge R(x) > R(x')$$

Definition 40 (Non-Termination Formula – Open Recurrence Set [**ONT**]).

$$\begin{aligned}\exists N, x_0.\forall x.\exists x'.N(x_0) \wedge& \\ &N(x) \rightarrow G(x) \wedge \\ &N(x) \rightarrow T(x,x') \wedge N(x')\end{aligned}$$

Definition 41 (Non-Termination Formula – Closed Recurrence Set [**CNT**]).

$$\begin{aligned}\exists N, x_0.\forall x, x'.N(x_0) \wedge& \\ &N(x) \rightarrow G(x) \wedge \\ &N(x) \wedge T(x,x') \rightarrow N(x')\end{aligned}$$

Definition 42 (Non-Termination Formula – Skolemized Open Recurrence Set [**SNT**]).

$$\begin{aligned}\exists N, C, x_0.\forall x.N(x_0) \wedge& \\ &N(x) \rightarrow G(x) \wedge \\ &N(x) \rightarrow T(x,C(x)) \wedge N(C(x))\end{aligned}$$

Figure 7.3: Formulae encoding the termination and non-termination of a single loop

As the existence of a ranking function is equivalent to the satisfiability of the formula [**UT**], a satisfiability witness is a ranking function and thus a proof of L's unconditional termination.

Returning to the program from Figure 7.2a, we can see that the corresponding second-order SAT formula [**UT**] is satisfiable, as witnessed by the function $R(x) = x$. Thus, $R(x) = x$ constitutes a proof that the program in Figure 7.2a is unconditionally terminating.

Note that different formulations for unconditional termination are possible. We are aware of a proof rule based on transition invariants, i.e. supersets of the transition relation's transitive closure [47]. This formulation assumes that the second-order logic has a primitive predicate for disjunctive well-foundedness. By contrast, our formulation in Definition 39 does not use a primitive disjunctive well-foundedness predicate.

Non-termination. Dually to termination, we might want to consider the non-termination of a loop. If a loop terminates, we can prove this by finding a ranking function witnessing the satisfiability of formula [**UT**]. What then would a proof of non-termination look like?

Since our program's state space is finite, a transition relation induces an infinite execution iff some state is visited infinitely often, or equivalently $\exists x.T^+(x, x)$. Deciding satisfiability of this formula directly would require a logic that includes a transitive closure operator, $\bullet^+$. Rather than introduce such an operator, we will characterise non-termination using the second-order SAT formula [**ONT**] (Definition 40, Figure 7.3) encoding the existence of an *(open) recurrence set*, i.e. a nonempty set of states N such that for each $s \in N$ there exists a transition to some $s' \in N$ [51].

Theorem 44. *The loop $L(G, T)$ has an infinite execution iff formula* [**ONT**] *(Definition 40) is satisfiable.*

If this formula is satisfiable, N is an open recurrence set for L, which proves L's non-termination. The issue with this formula is the additional level of quantifier alternation as compared to second-order SAT (it is an $\exists\forall\exists$ formula). To eliminate the innermost existential quantifier, we introduce a Skolem function C that chooses the successor x', which we then existentially quantify over. This results in formula [**SNT**] (Definition 42, Figure 7.3).

Theorem 45. *Formula* [**ONT**] *(Definition 40) and formula* [**SNT**] *(Definition 42) are equisatisfiable.*

This extra second-order term introduces some complexity to the formula, which we can avoid if the transition relation T is deterministic.

Definition 46 (Determinism)**.** A relation T is deterministic iff each state x has exactly one successor under T:

$$\forall x.\exists x'.T(x,x') \wedge \forall x''.T(x,x'') \rightarrow x'' = x'$$

In order to describe a deterministic *program* in a way that still allows us to sensibly talk about termination, we assume the existence of a special sink state s with no outgoing transitions and such that $\neg G(s)$ for any of the loop guards G. The program is deterministic if its transition relation is deterministic for all states except s.

When analysing a deterministic loop, we can make use of the notion of a *closed recurrence set* introduced by Chen et al. in [26]: for each state in the recurrence set N, *all* of its successors must be in N. The existence of a closed recurrence set is equivalent to the satisfiability of formula [**CNT**] in Definition 41, which is already in second-order SAT without needing Skolemization.

We note that if T is deterministic, every open recurrence set is also a closed recurrence set (since each state has at most one successor). Thus, the non-termination problem for deterministic transition systems is equivalent to the satisfiability of formula [**CNT**] from Figure 7.3.

Theorem 47. *If T is deterministic, formula* [**ONT**] *(Definition 40) and formula* [**CNT**] *(Definition 41) are equisatisfiable.*

So if our transition relation is deterministic, we can say, without loss of generality, that non-termination of the loop is equivalent to the existence of a closed recurrence set. However if T is non-deterministic, it may be that there is an open recurrence set but not closed recurrence set. To see this, consider the following loop:

```
while(x != 0) {
  y = nondet();
  x = x-y;
}
```

It is clear that this loop has many non-terminating executions, e.g. the execution where nondet() always returns 0. However each state has a successor that exits the loop, i.e. when nondet() returns the value currently stored in x. So this loop has an open recurrence set, but no closed recurrence set and hence we cannot give a proof of its non-termination with [**CNT**] and instead must use [**SNT**].

```
L1 :
while (i<n){
  j = 0;

L2 :
  while (j≤i){
    j = j + 1;
  }

  i = i + 1;
}
```

$$\begin{aligned}
&\exists T_o, R_1, R_2. \forall i, j, n, i', j', n'.\\
&\qquad i < n \rightarrow T_o(\langle i, j, n\rangle, \langle i, 0, n\rangle) \wedge\\
&j \le i \wedge T_o(\langle i', j', n'\rangle, \langle i, j, n\rangle) \rightarrow R_2(i, j, n) > 0 \wedge\\
&\qquad R_2(i, j, n) > R_2(i, j+1, n) \wedge\\
&\qquad T_o(\langle i', j', n'\rangle, \langle i, j+1, n\rangle) \wedge\\
&i < n \wedge S(\langle i, j, n\rangle, \langle i', j', n'\rangle) \wedge j' > i' \rightarrow R_1(i, j, n) > 0 \wedge\\
&\qquad R_1(i, j, n) > R_1(i+1, j, n)
\end{aligned}$$

Figure 7.4: A program with nested loops and its termination formula

Definition 48 (Conditional Termination Formula [**CT**]).

$$\begin{aligned}
&\exists R, W. \forall x, x'. I(x) \wedge G(x) \rightarrow W(x) \wedge\\
&\qquad G(x) \wedge W(x) \wedge T(x, x') \rightarrow W(x') \wedge R(x) > 0 \wedge R(x) > R(x')
\end{aligned}$$

Figure 7.5: Formula encoding conditional termination of a loop

7.6.2 An Isolated, Nested Loop

Termination. If a loop $L(G, T)$ has another loop $L'(G', T')$ nested inside it, we cannot directly use [**UT**] to express the termination of L. This is because the single-step transition relation T must include the transitive closure of the inner loop T'^*, and we do not have a transitive closure operator in our logic. Therefore to encode the termination of L, we construct an over-approximation $T_o \supseteq T$ and use this in formula [**UT**] to specify a ranking function. Rather than explicitly construct T_o using, for example, abstract interpretation, we add constraints to our formula that encode the fact that T_o is an over-approximation of T, and that it is precise enough to show that R is a ranking function.

As the generation of such constraints is standard and covered by several other works [47, 50], we will not provide the full algorithm, but rather illustrate it through the example in Figure 7.4. For the current example, the termination formula is given on the right side of Figure 7.4: T_o is a summary of L_1 that over-approximates its transition relation; R_1 and R_2 are ranking functions for L_1 and L_2, respectively.
Non-Termination. Dually to termination, when proving non-termination, we need to under-approximate the loop's body and apply formula [**CNT**]. Under-approximating

```
L_1:
while (G_1) {
  P_1;

L_2:
  while (G_2) {
    B_2;
  }

  P_2;
}
```

$$
\begin{aligned}
\exists N_1, N_2, x_0. \forall x, x'. \\
N_1(x_0) \wedge \\
N_1(x) \rightarrow G_1(x) \wedge \\
N_1(x) \wedge P_1(x, x') \rightarrow N_2(x') \wedge \\
G_2(x) \wedge N_2(x) \wedge B_2(x, x') \rightarrow N_2(x') \wedge \\
\neg G_2(x) \wedge N_2(x) \wedge P_2(x, x') \rightarrow N_1(x')
\end{aligned}
$$

Figure 7.6: Formula encoding non-termination of nested loops

the inner loop can be done with a nested existential quantifier, resulting in $\exists\forall\exists$ alternation, which we could eliminate with Skolemization. However, we observe that unlike a ranking function, the defining property of a recurrence set is *non relational* – if we end up in the recurrence set, we do not care exactly where we came from as long as we know that it was also somewhere in the recurrence set. This allows us to cast non-termination of nested loops as the formula shown in Figure 7.6, which does not use a Skolem function.

If the formula on the right-hand side of the figure is satisfiable, then L_1 is non-terminating, as witnessed by the recurrence set N_1 and the initial state x_0 in which the program begins executing. There are two possible scenarios for L_2's termination:

- If L_2 is terminating, then N_2 is an inductive invariant that reestablished N_1 after L_2 stops executing: $\neg G_2(x) \wedge N_2(x) \wedge P_2(x, x') \rightarrow N_1(x')$.
- If L_2 is non-terminating, then $N_2 \wedge G_2$ is its recurrence set.

7.6.3 Composing a Loop with the Rest of the Program

Sometimes the termination behaviour of a loop depends on the rest of the program. That is to say, the loop may not terminate if started in some particular state, but that state is not actually reachable on entry to the loop. The program as a whole terminates, but if the loop were considered in isolation we would not be able to prove that it terminates. We must therefore encode a loop's interaction with the rest of the program in order to do a sound termination analysis.

Let us assume that we have done some preprocessing of our program which has identified loops, straight line code blocks and the control flow between these. In particular, the control flow analysis has determined which order these code blocks execute in, and the nesting structure of the loops.

Conditional termination. Given a loop $L(G, T)$, if L's termination depends on the state it begins executing in, we say that L is *conditionally terminating.* The information we require of the rest of the program is a predicate I which over-approximates the set of states that L may begin executing in. That is to say, for each state x that is reachable on entry to L, we have $I(x)$.

Theorem 49. *The loop $L(G, T)$ terminates when started in any state satisfying $I(x)$ iff formula* [**CT**] *(Definition 48, Figure 7.5) is satisfiable.*

If formula [**CT**] is satisfiable, two witnesses are returned:

- W is an inductive invariant of L that is established by the initial states I if the loop guard G is met.
- R is a ranking function for L as restricted by W – that is to say, R need only be well founded on those states satisfying $W \wedge G$. Since W is an inductive invariant of L, R is strong enough to show that L terminates from any of its initial states.

W is called a *supporting invariant* for L and R proves termination relative to W. We require that $I \wedge G$ is strong enough to establish the base case of W's inductiveness.

Conditional termination is illustrated by the program in Figure 7.2h, which is encoded as:

$$I(\langle x, y\rangle) \triangleq y = 1$$
$$G(\langle x, y\rangle) \triangleq x > 0$$
$$T(\langle x, y\rangle, \langle x', y'\rangle) \triangleq x' = x - y \wedge y' = y$$

If the initial states I are ignored, this loop cannot be shown to terminate, since any state with $y = 0$ and $x > 0$ would lead to a non-terminating execution.

However, formula [**CT**] is satisfiable, as witnessed by:

$$R(\langle x, y\rangle) = x$$
$$W(\langle x, y\rangle) \triangleq y = 1$$

This constitutes a proof that the program as a whole terminates, since the loop always begins executing in a state that guarantees its termination.

Definition 50 (Safety Invariant Formula [**SI**]).

$$\begin{aligned}\exists S.\forall x, x'.I(x) \rightarrow S(x) \wedge \\ S(x) \wedge G(x) \wedge B(x, x') \rightarrow S(x') \wedge \\ S(x) \wedge \neg G(x) \rightarrow A(x)\end{aligned}$$

Figure 7.7: Existence of a safety invariant as second-order SAT

7.7 Safety and Danger Proofs

7.7.1 Safety Invariants

A safety invariant is a set of states S which is inductive with respect to the program's transition relation, and which excludes an error state. A predicate S is a safety invariant for the loop I, G, B, A iff it satisfies the following criteria:

$$\forall x.I(x) \rightarrow S(x) \tag{7.1}$$

$$\forall x, x'.S(x) \wedge G(x) \wedge B(x, x') \rightarrow S(x') \tag{7.2}$$

$$\forall x.S(x) \wedge \neg G(x) \rightarrow A(x) \tag{7.3}$$

7.1 says that each state reachable on entry to the loop is in the set S, and in combination with 7.2 shows that every state that can be reached by the loop is in S. The final criterion 7.3 says that if the loop exists while in an S-state, the assertion A is not violated. The existence of a safety invariant corresponds to the notion of partial correctness: no assertion will fail, but the program may never stop running. If we allow ourselves to quantify over predicates, we can specify the existence of a safety invariant with [**SI**] from Definition 50. If we restrict the language of predicates to $\mathcal{L}$-expressions and the relations I, G, B, A to C$^-$ programs, Definition 50 is a second-order SAT formula that we can solve directly with Kalashnikov.

7.7.2 Danger Invariants

Dually to safety invariants, danger invariants are underapproximations of the reachable states that guarantee an error will occur. Like safety invariants, we require that a danger invariant is inductive with respect to the loop, and that it holds in some initial state, although it need not hold in every initial state. A predicate D is a danger invariant for the loop I, G, B, A iff:

$$\exists x . I(x) \wedge D(x) \tag{7.4}$$

$$\forall x . D(x) \wedge G(x) \rightarrow \exists x' . B(x, x') \wedge D(x') \tag{7.5}$$

$$\forall x . D(x) \wedge \neg G(x) \rightarrow \neg A(x) \tag{7.6}$$

Similarly to the definition of a safety invariant, 7.4 says that there is some D-state in which the loop can begin executing. For the induction, 7.5 says that each D-state can reach at least one other D-state via an iteration of the loop. Finally 7.6 says that if the loop exits while in a D-state, the assertion fails. These properties are shown directly in the second-order SAT formula [**DI**] of Definition 51. The existence of a danger invariant shows that if the loop exits having started in a D-state, an assertion will certainly fail. As with the non-termination formula [**ONT**], we must eliminate the innermost existential quantifier from [**DI**]. If the transition relation is deterministic, we can use formula [**DDI**] (again by observing that each state has exactly one successor). Otherwise, we must Skolemise, giving us [**SDI**].

The satisfiability of one of the danger formulae is not quite enough for us to conclude that an assertion *does* fail, since we have not yet established that the loop must terminate from any D-state – we have a notion of partial-incorrectness, where we would like total-incorrectness. In order to guarantee that there is a trace through the program, via our danger invariant which certainly terminates in an error state, we must introduce a ranking function. This is done in formula [**TDI**] of Definition 54 – the ranking function R ensures that each iteration of the loop makes progress towards a state in which the assertion A is violated. As with [**DI**], we can eliminate the innermost existential quantifier from [**TDI**], by replacing it with a universal if the program under analysis is deterministic, or by Skolemisation otherwise.

We have finally arrived at a second-order SAT formula which exactly captures danger for a program – formula [**TDI**] is true iff there is a finite trace through our program ending with a violation of the assertion A.

7.8 Analysing Programs with Second-Order Tautologies

The second-order SAT solver described in Chapter 6 is efficient at finding satisfying assignments to true formulae, when such a solution has low Kolmogorov complexity. However if a formula is unsatisfiable, the procedure will not terminate in practice.

Definition 51 (Danger Invariant Formula [**DI**]).

$$\begin{aligned}\exists D, x_0 . \forall x . \exists x' . I(x_0) \wedge D(x_0) \wedge \\ D(x) \wedge G(x) \rightarrow B(x, x') \wedge D(x') \wedge \\ D(x) \wedge \neg G(x) \rightarrow \neg A(x)\end{aligned}$$

Definition 52 (Deterministic Danger Invariant Formula [**DDI**]).

$$\begin{aligned}\exists D, x_0 . \forall x, x' . I(x_0) \wedge D(x_0) \wedge \\ D(x) \wedge G(x) \wedge B(x, x') \rightarrow D(x') \wedge \\ D(x) \wedge \neg G(x) \rightarrow \neg A(x)\end{aligned}$$

Definition 53 (Skolemized Danger Invariant Formula [**SDI**]).

$$\begin{aligned}\exists D, S, x_0 . \forall x . I(x_0 \wedge D(x_0) \wedge \\ D(x) \wedge G(x) \rightarrow B(x, S(x)) \wedge D(S(x)) \wedge \\ D(x) \wedge \neg G(x) \rightarrow \neg A(x)\end{aligned}$$

Definition 54 (Total Danger Formula [**TDI**]).

$$\begin{aligned}\exists D, x_0, R . \forall x . \exists x' . I(x_0) \wedge D(x_0) \wedge \\ D(x) \wedge G(x) \rightarrow R(x) > 0 \wedge B(x, x') \wedge D(x') \wedge R(x) > R(x') \wedge \\ D(x) \wedge \neg G(x) \rightarrow \neg A(x)\end{aligned}$$

Figure 7.8: Existence of a danger invariant as second-order SAT

We are therefore interested in finding formulae encoding interesting properties of a program, such that we know the formula will be satisfiable. While there is no apparent value in checking satisfiability of a formula we know to be satisfiable, we will show in this section that the approach does make sense in some scenarios.

Let us consider termination. We have some loop L and we would like to know if it terminates or not. In Section 7.6 we derived second-order SAT formulae that exactly characterised termination and non-termination for an arbitrary loop L. In particular, formula [**CT**] from Definition 48 is true iff L terminates from every initial state, and formula [**ONT**] from Definition 40 is true iff L loops from some initial state. It is clear that exactly one of these formulae is true. Since both are second-order SAT formulae (i.e. the second-order terms are existentially quantified – we haven't generated [**ONT**] just by negating [**CT**]), we can take their disjunction. Let's say that the particular instantiations of [**CT**] and [**ONT**] for L are respectively:

$$\exists P_T.\forall x, x'.\phi(P_T, x, x')$$
$$\exists P_N.\forall x.\psi(P_n, x)$$

the disjunction is then:

$$(\exists P_T.\forall x, x'.\phi(P_T, x, x')) \vee (\exists P_N.\forall x.\, \psi(P_N, x))$$

Which, after renaming bound variables, simplifies to:

$$\exists P_T, P_N.\forall x, x', y.\, \phi(P_T, x, x') \vee \psi(P_N, y)$$

From which we arrive at the generalised termination formula of Definition 55, [**GT**]. Assuming a (non-)termination proof for L is expressible in $\mathcal{L}$, exactly one of [**CT**] and [**NT**] is true. Therefore [**GT**] is a theorem of second-order logic for any loop L whose termination is provable in $\mathcal{L}$. We will show in Section 7.9 that in fact *every* loop has a termination proof expressible in $\mathcal{L}$, and so [**GT**] is always a theorem. A solution to the formula would include witnesses P_N and P_T, which are putative proofs of non-termination and termination respectively. Exactly one of these will be a genuine proof, so we can check first one and then the other, which requires a single call to a SAT solver for each check.

We can perform the same trick for safety, which gives us the generalised safety formula [**GS**] of Definition 56. This is the disjunction of [**SI**], which encodes partial

Definition 55 (Generalised Termination Formula [**GT**]).

$$\exists R, W, N, y_0 . \forall x, x', y . \exists y' . \begin{pmatrix} I(x) \wedge G(x) \rightarrow W(x) \wedge \\ G(x) \wedge W(x) \wedge B(x, x') \rightarrow W(x') \wedge R(x) > 0 \wedge R(x) > R(x') \end{pmatrix} \vee \begin{pmatrix} I(y_0) \wedge N(y_0) \wedge \\ N(y) \rightarrow G(y) \wedge \\ N(y) \rightarrow B(y, y') \wedge N(y') \end{pmatrix}$$

Definition 56 (Generalised Safety Formula [**GS**]).

$$\exists S, D, R, y_0 . \forall x, x', y . \exists y' . \begin{pmatrix} I(x) \rightarrow S(x) \wedge \\ S(x) \wedge G(x) \wedge B(x, x') \rightarrow S(x') \wedge \\ S(x) \wedge \neg G(x) \rightarrow A(x) \end{pmatrix} \vee \begin{pmatrix} I(y_0) \wedge D(y_0) \wedge \\ D(y) \wedge G(y) \rightarrow B(y, y') \wedge D(y') \wedge R(y) > 0 \wedge R(y) > R(y') \wedge \\ D(y) \wedge \neg G(y) \rightarrow \neg A(y) \end{pmatrix}$$

Figure 7.9: General second-order SAT formula characterising safety

correctness, and [**TDI**], which encodes total danger. Again, if a safety proof is expressible in $\mathcal{L}$, exactly one of these formulae will be satisfiable and so [**GS**] will be a theorem.

Note that we can only generate a second-order tautology for some particular program property if both the property *and its dual* are encodable as second-order SAT formulae. We have shown that this is the case for both termination and safety, but since second-order SAT is not closed under complement, this need not be the case for every property.

7.9 Soundness, Completeness and Complexity

In this section, we show that $\mathcal{L}$ is expressive enough to capture (non-)termination proofs for every bit-vector program. Using this result, we then show that our analysis terminates with a valid (non-)termination proof for every input program.

Lemma 57. *Every finite subset $A \subseteq B$ is computable by a finite $\mathcal{L}$-program by setting $X = B, Y = 2$ in Lemma 25 and taking the resulting function to be the characteristic function of A.*

Theorem 58. *Every terminating bit-vector program has a ranking function that is expressible in $\mathcal{L}$.*

Proof. Let $v_1, \ldots, v_k$ be the variables of the program P under analysis, and let each be b bits wide. Its state space $\mathcal{S}$ is then of size 2^{bk}. A ranking function $R : \mathcal{S} \to \mathcal{D}$ for P exists iff P terminates. Without loss of generality, $\mathcal{D}$ is a well-founded total order. Since R is injective, we have that $|\mathcal{D}| \geq |\mathcal{S}|$. If $|\mathcal{D}| > |\mathcal{S}|$, we can construct a function $R' : \mathcal{S} \to \mathcal{D}'$ with $|\mathcal{D}'| = |\mathcal{S}|$ by just setting $R' = R|_{\mathcal{S}}$, i.e. R' is just the restriction of R to $\mathcal{S}$. Since $\mathcal{S}$ already comes equipped with a natural well ordering we can also construct $R'' = \iota \circ R'$ where $\iota : \mathcal{D}' \to \mathcal{S}$ is the unique order isomorphism from $\mathcal{D}'$ to $\mathcal{S}$. So assuming that P terminates, there is some ranking function R'' that is just a permutation of $\mathcal{S}$. If the number of variables $k > 1$ then in general the ranking function will be lexicographic with dimension $\leq k$ and each co-ordinate of the output being a single b-bit value.

Then by Lemma 25 with $X = Y = \mathcal{S}$, there exists a finite $\mathcal{L}$-program computing R''. □

Theorem 59. *Every non-terminating bit-vector program has a non-termination proof expressible in $\mathcal{L}$.*

Proof. A proof of non-termination is a triple $\langle N, C, x_0 \rangle$ where $N \subseteq \mathcal{S}$ is a (finite) recurrence set and $C : \mathcal{S} \to \mathcal{S}$ is a Skolem function choosing a successor for each $x \in N$. $\mathcal{S}$ is finite, so by Lemma 25 both N and C are computed by finite $\mathcal{L}$-programs and x_0 is just a ground term. □

Theorem 60. *The generalised termination formula* [**GT**] *for any loop L is a theorem when P_N and P_T range over $\mathcal{L}$-computable functions.*

Proof. For any P, P', σ, σ, if $P \models \sigma$ then $(P, P') \models \sigma \lor \sigma'$.

By Theorem 58, if L terminates then there exists a termination proof P_T expressible in $\mathcal{L}$. Since ϕ is an instance of [**CT**], $P_T \models \phi$ (Theorem 49) and for any P_N, $(P_T, P_N) \models \phi \lor \psi$.

Similarly if L does not terminate for some input, by Theorem 59 there is a non-termination proof P_N expressible in $\mathcal{L}$. Formula ψ is an instance of [**SNT**] and so $P_N \models \psi$ (Theorem 45), hence for any P_T, $(P_T, P_N) \models \phi \lor \psi$.

So in either case (L terminates or does not), there is a witness in $\mathcal{L}$ satisfying $\phi \lor \psi$, which is an instance of [**GT**]. □

Theorem 61. *Our termination analysis is sound and complete – it terminates for all input loops L with a correct termination verdict.*

Proof. By Theorem 60, the specification spec is satisfiable. In [64], we show that the second-order SAT solver is semi-complete, and so is guaranteed to find a satisfying assignment for spec. If L terminates then P_T is a termination proof (Theorem 49), otherwise P_N is a non-termination proof (Theorem 45). Exactly one of these purported proofs will be valid, and since we can check each proof with a single call to a SAT solver we simply test both and discard the one that is invalid. □

7.10 Experiments

To evaluate our termination algorithm, we implemented a tool that generates a termination specification from a C program and calls the second-order SAT solver in [64] to obtain a proof. We ran the resulting termination prover, named JUGGERNAUT, on 47 benchmarks taken from the literature and SV-COMP'15 [96]. We omitted exactly those SVCOMP'15 benchmarks that made use of arrays or recursion. We do not have arrays in our logic and we had not implemented recursion in our frontend (although the latter can be syntactically rewritten to our input format).

To provide a comparison point, we also ran ARMC [89] on the same benchmarks. Each tool was given a time limit of 180 s, and was run on an unloaded 8-core 3.07 GHz Xeon X5667 with 50 GB of RAM. The results of these experiments are given in Figure 7.10.

It should be noted that the comparison here is imperfect, since ARMC is solving a different problem – it checks whether the program under analysis would terminate if run with unbounded integer variables, while we are checking whether the program terminates with bit-vector variables. This means that ARMC's verdict differs from ours in 3 cases (due to the differences between integer and bit-vector semantics). There are a further 7 cases where our tool is able to find a proof and ARMC cannot, which we believe is due to our more expressive proof language. In 3 cases, ARMC times out while our tool is able to find a termination proof. Of these, 2 cases have nested loops and the third has an infinite number of terminating lassos. This is not a problem for us, but can be difficult for provers that enumerate lassos.

On the other hand, ARMC is *much* faster than our tool. While this difference can partly be explained by much more engineering time being invested in ARMC, we feel that the difference is probably inherent to the difference in the two approaches – our solver is more general than ARMC, in that it provides a complete proof system for both termination and non-termination. This comes at the cost of efficiency: JUGGERNAUT is slow, but unstoppable.

Of the 47 benchmarks, 2 use nonlinear operations in the program (loop6 and loop11), and 5 have nested loops (svcomp6, svcomp12, svcomp18, svcomp40, svcomp41). JUGGERNAUT handles the nonlinear cases correctly and rapidly. It solves 4 of the 5 nested loops in less than 30 s, but times out on the 5th.

In conclusion, these experiments confirm our conjecture that second-order SAT can be used effectively to prove termination and non-termination. In particular, for programs with nested loops, nonlinear arithmetic and complex termination arguments, the versatility given by a general purpose solver is very valuable.

7.11 Related Work

There has been substantial prior work on automated program termination analysis. Figure 7.1 summarises the related work with respect to the assumptions they make about programs and ranking functions. Most of the techniques are specialised in the synthesis of linear ranking functions for linear programs over integers (or rationals) [8, 19, 32, 33, 54, 68, 75, 86]. Among them, Lee et al. make use of transition predicate abstraction, algorithmic learning, and decision procedures [75], Leike and Heizmann propose linear ranking templates [76], whereas Bradley et al. compute lexicographic linear ranking functions supported by inductive linear invariants [19].

While the synthesis of termination arguments for linear programs over integers is indeed well covered in the literature, there is very limited work for programs over machine integers. Cook et al. present a method based on a reduction to Presburger arithmetic, and a template-matching approach for predefined classes of ranking functions based on reduction to SAT- and QBF-solving [31]. Similarly, the only work we are aware of that can compute nonlinear ranking functions for imperative loops with polynomial guards and polynomial assignments is [21]. However, this work extends only to polynomials.

Given the lack of research on termination of nonlinear programs, as well as programs over bit-vectors and floats, our work focused on covering these areas. One of the obvious conclusions that can be reached from Figure 7.1 is that most methods tend to specialise on a certain aspect of termination proving that they can solve efficiently. Conversely to this view, we aim for generality, as we do not restrict the form of the synthesised ranking functions, nor the form of the input programs.

As mentioned in Section 7.2, approaches based on Ramsey's theorem compute a set of local termination conditions that decrease as execution proceeds through the loop and require expensive reachability analyses [30, 32, 87]. In an attempt to reduce the

		ARMC		JUGGERNAUT	
Benchmark	Expected	Verdict	Time	Verdict	Time
loop1.c	✓	✓	0.06s	✓	1.3s
loop2.c	✓	✓	0.06s	✓	1.4s
loop3.c	✓	✓	0.06s	✓	1.8s
loop4.c	✓	✓	0.12s	✓	2.9s
loop5.c	✓	✓	0.12s	✓	5.3s
loop6.c	✓	✓	0.05s	✓	1.2s
loop7.c [25]	✓	?	0.05s	✓	8.3s
loop8.c	✓	?	0.06s	✓	1.3s
loop9.c	✓	✓	0.11s	✓	1.6s
loop10.c	✓	✗	0.05s	✓	1.3s
loop11.c	✗	✓	0.05s	✗	1.4s
loop43.c [33]	✓	✓	0.07s	✓	1.5s
loop44.c [33]	✗	?	0.05s	✗	10.5s
loop45.c [33]	✓	✓	0.12s	✓	4.3s
loop46.c [33]	✓	?	0.05s	✓	1.5s
loop47.c	✓	✓	0.10s	✓	1.8s
loop48.c	✓	✓	0.06s	✓	1.4s
loop49.c	✗	?	0.05s	✗	1.3s
svcomp1.c [4]	✓	✓	0.11s	✓	2.3s
svcomp2.c	✓	✓	0.05s	✓	1.5s
svcomp3.c [7]	✓	✓	0.15s	✓	146.4s
svcomp4.c [19]	✗	✗	0.09s	✗	2.1s
svcomp5.c [20]	✓	✓	0.38s	–	T/O
svcomp6.c [25]	✓	–	T/O	✓	29.1s
svcomp7.c [25]	✓	✓	0.09s	✓	5.5s
svcomp8.c [27]	✓	?	0.05s	–	T/O
svcomp9.c [33]	✓	✓	0.10s	✓	1.5s
svcomp10.c [33]	✓	✓	0.11s	✓	4.5s
svcomp11.c [33]	✓	✓	0.20s	✓	14.6s
svcomp12.c [39]	✓	–	T/O	✓	10.9s
svcomp13.c	✓	?	0.07s	✓	35.1s
svcomp14.c [48]	✓	–	T/O	✓	30.8s
svcomp15.c [53]	✓	?	0.12s	–	T/O
svcomp16.c [53]	✓	✓	0.06s	✓	2.2s
svcomp17.c [68]	✓	✓	0.05s	–	T/O
svcomp18.c [74]	✓	?	0.27s	–	T/O
svcomp25.c	✓	?	0.05s	–	T/O
svcomp26.c	✓	✓	0.26s	✓	3.2s
svcomp27.c [84]	✗	✓	0.11s	–	T/O
svcomp28.c [84]	✓	✓	0.13s	–	T/O
svcomp29.c [86]	✓	?	0.05s	–	T/O
svcomp37.c	✓	✓	0.16s	✓	2.1s
svcomp38.c	✓	✓	0.10s	–	T/O
svcomp39.c	✓	✓	0.25s	–	T/O
svcomp40.c [98]	✓	?	0.07s	✓	25.5s
svcomp41.c [98]	✓	?	0.07s	✓	25.5s
svcomp42.c	✓	✓	0.22s	–	T/O
Correct			28		35
Incorrect for bit-vectors			3		0
Unknown			13		0
Timeout			3		12

Key: ✓= terminating, ✗= non-terminating, ? = unknown (tool terminated with an inconclusive verdict).

Figure 7.10: Experimental results

complexity of checking the validity of the termination argument, Cook et al. present an iterative termination proving procedure that searches for lexicographic termination arguments [33], whereas Kroening et al. strengthen the termination argument such that it becomes a transitive relation [68]. Following the same trend, we search for lexicographic nonlinear termination arguments that can be verified with a single call to a SAT solver.

Proving program termination implies the simultaneous search for a termination argument and a supporting invariant. Brockschmidt et al. share the same representation of the state of the termination proof between the safety prover and the ranking function synthesis tool [25]. Bradley et al. combine the generation of ranking functions with the generation of invariants to form a single constraint solving problem such that the necessary supporting invariants for the ranking function are discovered on demand [19]. In our setting, both the ranking function and the supporting invariant are iteratively constructed in the same refinement loop.

While program termination has been extensively studied, much less research has been conducted in the area of proving non-termination. Gupta et al. dynamically enumerate lasso-shaped candidate paths for counterexamples, and then statically prove their feasibility [51]. Chen et al. prove non-termination via reduction to safety proving [26]. Their iterative algorithm uses counterexamples to a fixed safety property to refine an under-approximation of a program. In order to prove both termination and non-termination, Harris et al. compose several program analyses (termination provers for multi-path loops, non-termination provers for cycles, and global safety provers) [53]. We propose a uniform treatment of termination and non-termination by formulating a generalised second-order formula whose solution is a proof of one of them.

Chapter 8

Propositional Reasoning About the Heap

8.1 Introduction

Proving safety of heap-manipulating programs is a notoriously difficult task. One of the main culprits is the complexity of the verification conditions generated for such programs. The constraints comprising these verification conditions can be arithmetic (e.g. the value stored at location pointed by x is equal to 3), structural (e.g. x points to an acyclic singly-linked list), or a combination of the first two when certain structural properties of a data structure are captured as numeric values (e.g. the length of the list pointed by x is 3). Solving these combined constraints requires non-trivial interaction between shape and arithmetic.

For illustration, consider the program in Figure 8.1b, which iterates simultaneously over the lists x and y. The program is safe, i.e. there is no null pointer dereferencing and the assertion after the loop holds. While the absence of null pointer dereferences is trivial to observe and prove, the fact that the assertion after the loop holds relies on the fact that at the beginning of the program and after each loop iteration the lengths of the lists z and t are equal. Thus, the specification language must be capable of expressing the fact that both z and t reach null in the same number of steps. Note that the interaction between shape and arithmetic constraints is intricate, and cannot be solved by a mere theory combination.

The problem is even more pronounced when proving termination of heap-manipulating programs. The reason is that, even more frequently than in the case of safety checking, termination arguments depend on the size of the heap data structures. For example, a loop iterating over the nodes of such a data structure terminates after all the reachable nodes have been explored. Thus, the termination argument is directly linked to the number of nodes in the data structure. This situation is illustrated again by the loop in Figure 8.1b.

There are few logics capable of expressing this type of interdependent shape and arithmetic constraint. One of the reasons is that, given the complexity of the constraints, such logics can easily become undecidable (even the simplest use of transitive closure leads to undecidability [59]), or at best inefficient.

The tricky part is identifying a logic that is expressive enough to capture the corresponding constraints and at the same time is efficiently decidable. One work that inspired us in this endeavour is the recent approach by Itzhaky et al. on reasoning about reachability between dynamically allocated memory locations in linked lists using effectively-propositional (EPR) reasoning [60]. This result is appealing as it can harness advances in SAT solvers. The only downside is that the logic presented in [60] is better suited for safety than termination checking, and is best for situations where safety does not depend on the interaction between shape and arithmetic. Thus, our goal is to define a logic that can be used in such scenarios while still being reducible to SAT.

This paper shows that it is possible to reason about the safety and termination of programs handling potentially cyclic, singly-linked lists using propositional reasoning. For this purpose, we present the logic SLH which can express interdependent shape and arithmetic constraints. We empirically show its utility for the verification of heap-manipulating programs by using it to express safety invariants and termination arguments for intricate programs with potentially cyclic, singly-linked lists with unrestricted, unspecified sharing.

SLH is parametrised by the background arithmetic theory used to express the length of lists (and implicitly every numeric variable). The decision procedure reduces validity of a formula in SLH to satisfiability of a formula in the background theory. Thus, SLH is decidable if the background theory is decidable.

As we are interested in a reduction to SAT, we instantiate SLH with the theory of bit-vector arithmetic, resulting in SLH[$\mathcal{T}_{\mathcal{BV}}$]. This allows us to handle non-linear operations on lists length (e.g. the example in Figure 8.1c), while still retaining decidability. However, SLH can be combined with other background theories, e.g. Presburger arithmetic.

We provide an implementation of our decision procedure for SLH[$\mathcal{T}_{\mathcal{BV}}$] and test its efficiency by verifying a suite of programs against safety and termination specifications expressed in SLH. Whenever the verification fails, our decision procedure produces a counterexample.

Contributions:

- We propose the theory SLH of singly-linked lists with length. SLH allows *unrestricted* sharing and cycles.
- We define the strongest post-condition for formulae in SLH.
- We show the utility of SLH for software verification by using it to express safety invariants and termination arguments for programs with potentially cyclic singly-linked lists.
- We present the instantiation SLH[$\mathcal{T}_{\mathcal{BV}}$] of SLH with the theory of bit-vector arithmetic. SLH[$\mathcal{T}_{\mathcal{BV}}$] can express non-linear operations on the lengths of lists, while still retaining decidability.
- We provide a reduction from satisfiability of SLH[$\mathcal{T}_{\mathcal{BV}}$] to propositional SAT.
- We provide an implementation of the decision procedure for SLH[$\mathcal{T}_{\mathcal{BV}}$] and test it by checking safety and termination for several heap-manipulating programs (against provided safety invariants and termination arguments).

8.2 Motivation

Consider the examples in Figure 8.1. They all capture situations where the safety (i.e. absence of null pointer dereferencing and no assertion failure) and termination of the program depend on interdependent shape and arithmetic constraints. In this section we only give an intuitive description of these examples, and we revisit and formally specify them in Section 8.7. We assume the existence of the following two functions: (1) $length(x)$ returns the number of nodes on the path from x to NULL if the list pointed by x is acyclic, and MAXINT otherwise; (2) $circular(x)$ returns true iff the list pointed by x is circular (i.e. x is part of a cycle).

In Figure 8.1a, we iterate over the potentially cyclic singly-linked list pointed by x a number of times equal with the result of $length(x)$. The program is safe (i.e. y is not NULL at loop entry) and terminating. A safety invariant for the loop needs to capture the length of the path from y to NULL.

The loop in Figure 8.1b iterates over the lists pointed by x and y, respectively, until one of them becomes NULL. In order to check whether the assertion after the loop holds, the safety invariant must relate the length of the list pointed by x to the

```
List x, y = x;
int n = length(x), i = 0;

while (i < n) {
  y = y→next;
  i = i+1;
}
```

(a)

```
List x, y, z = x, t = y;

assume(length(x) == length(y));

while (z != NULL && t != NULL) {
  z = z→next;
  t = t→next;
}

assert (z == NULL && t == NULL);
```

(b)

```
int divides(List x, List y) {
  List z = y;
  List w = x;

  assume(length(x) != MAXINT &&
         length(y) != MAXINT &&
         y != NULL);

  while (w != NULL) {
    if (z == NULL) z = y;
    z = z→next;
    w = w→next;
  }

  assert(z == NULL ⇔
         length(x)%length(y) == 0);
  return z == NULL;
}
```

(c)

```
int isCircular(List l) {
  List p = q = l;

  do {
    if (p != NULL) p = p→next;
    if (q != NULL) q = q→next;
    if (q != NULL) q = q→next;
  }
  while (p != NULL &&
         q != NULL &&
         p != q);

  assert(p == q ⇔ circular(l));
  return p == q;
}
```

(d)

Figure 8.1: Motivational examples.

length of the list pointed by y. Similarly, a termination argument needs to consider the length of the two lists.

The example in Figure 8.1c illustrates how non-linear arithmetic can be encoded via singly-linked lists. Thus, the loop in $\mathit{divides}(x, y)$ iterates over the list pointed by x a number of nodes equal to the quotient of the integer division $\mathit{length}(x)/\mathit{length}(y)$ such that, after the loop, the list pointed by z has a length equal with the remainder of the division.

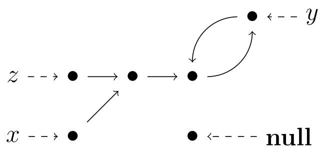
The function in Figure 8.1d returns true iff the list passed in as a parameter is circular. The functional correctness of this function is captured by the assertion after the loop checking that pointers p and q end up being equal iff the list l is circular.

8.3 Theory of Singly Linked Lists with Length

In this section we introduce the theory SLH for reasoning about potentially cyclic singly linked lists.

8.3.1 Informal Description of SLH

We imagine that there is a set of pointer variables $x, y, \ldots$ which point to heap cells. The cells in the heap are arranged into singly linked lists, i.e. each cell has a "next" pointer which points somewhere in the heap. The lists can be cyclic and two lists can share a tail, so for example the following heap is allowed in our logic:



Our logic contains functions for examining the state of the heap, along with the four standard operations for mutating linked lists: *new*, *assign*, *lookup* and *update*. We capture the side-effects of these mutation operators by explicitly naming the current heap – we introduce heap variables h, h' etc. which denote the heap in which each function is to be interpreted. The mutation operators then become pure functions mapping heaps to heaps. The heap functions of the logic are illustrated by example in Figure 8.2 and have the following meanings:

$\mathit{alias}(h, x, y)$:	do x and y point to the same cell in heap h?
$\mathit{isPath}(h, x, y)$:	is there a path from x to y in h?
$\mathit{pathLength}(h, x, y)$:	the length of the shortest path from x to y in h.
$\mathit{isNull}(h, x)$:	is x **null** in h?
$\mathit{circular}(h, x)$:	is x part of a cycle, i.e. is there some non-empty path from x back to x in h?
$h' = \mathit{new}(h, x)$:	obtain h' from h by allocating a new heap cell and reassigning x so that it points to this cell. The newly allocated cell is not reachable from any other cell and its successor is **null**. This models the program statement $x = \mathit{new}()$. For simplicity, we opt for this allocation policy, but we are not restricted to it.
$h' = \mathit{assign}(h, x, y)$:	obtain h' from h by assigning x so that it points to the same cell as y. Models the statement $x = y$.
$h' = \mathit{lookup}(h, x, y)$:	obtain h' from h by assigning x to point to y's successor. Models the statement $x = y \rightarrow \mathit{next}$.
$h' = \mathit{update}(h, x, y)$:	obtain h' from h by updating x's successor to point to y. Models $x \rightarrow \mathit{next} = y$.

8.3.2 Syntax of SLH

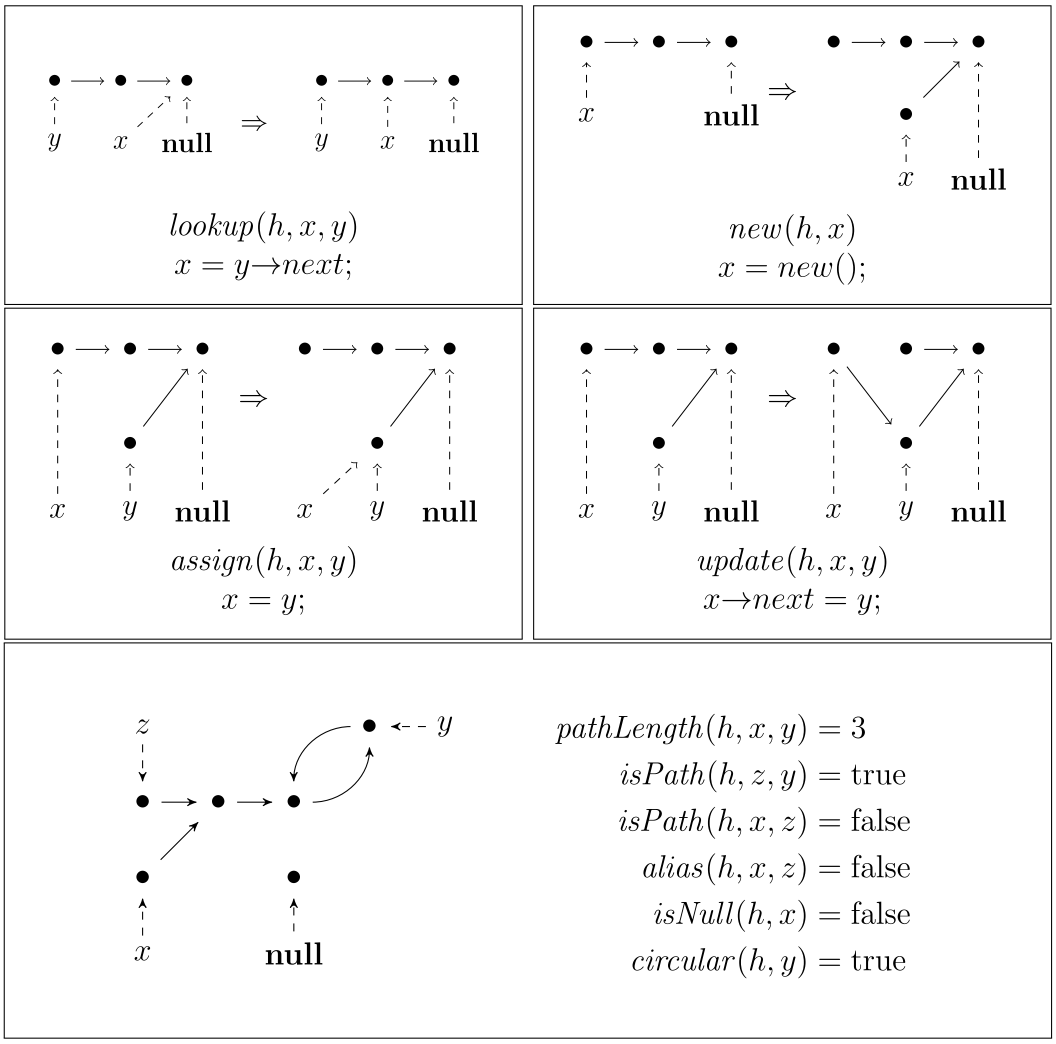
The theory of singly-linked lists with length, SLH, uses a background arithmetic theory $\mathcal{T_B}$ for the length of lists (implicitly any numeric variable). Thus, SLH has the following signature:

$$\begin{aligned}\Sigma_{SLH} = \quad & \Sigma_B \cup \{\mathit{alias}(\cdot,\cdot,\cdot), \mathit{isPath}(\cdot,\cdot,\cdot), \mathit{isNull}(\cdot,\cdot), \mathit{circular}(\cdot,\cdot),\\ & \mathit{pathLength}(\cdot,\cdot,\cdot), \cdot{=}\mathit{new}(\cdot,\cdot), \cdot{=}\mathit{assign}(\cdot,\cdot,\cdot),\\ & \cdot{=}\mathit{lookup}(\cdot,\cdot,\cdot), \cdot{=}\mathit{update}(\cdot,\cdot,\cdot)\}.\end{aligned}$$

where the nine new symbols correspond to the heap-specific functions described in the previous section (the first four are actually heap predicates).

Sorts. Heap variables (e.g. h in $\mathit{alias}(h, x, y)$) have sort $\mathcal{S_H}$, pointer variables have sort $\mathcal{S}_{\mathit{Addr}}$ (e.g. x and y in $\mathit{alias}(h, x, y)$), numeric variables have sort $\mathcal{S_B}$ (e.g. n in $n = \mathit{pathLength}(h, x, y)$).

Literal and formula. A literal in SLH is either a heap function (including the negation of the predicates) or a $\mathcal{T_B}$-literal (which may refer to $\mathit{pathLength}$). A formula in SLH is a Boolean combination of SLH-literals.



y
x
null
⇒
$lookup(h, x, y)$
$x = y{\to}next;$
x
null
$new(h, x)$
$x = new();$
$assign(h, x, y)$
$x = y;$
$update(h, x, y)$
$x{\to}next = y;$
z
y
x
null
$pathLength(h, x, y) = 3$
$isPath(h, z, y) = \text{true}$
$isPath(h, x, z) = \text{false}$
$alias(h, x, z) = \text{false}$
$isNull(h, x) = \text{false}$
$circular(h, y) = \text{true}$

Figure 8.2: SLH by example

8.3.3 Semantics of SLH

We give the semantics of SLH by defining the models in which an SLH formula holds. An interpretation Γ is a function mapping free variables to elements of the appropriate sort. If an SLH formula ϕ holds in some interpretation Γ, we say that Γ *models* ϕ and write $\Gamma \models \phi$.

Interpretations may be constructed using the following substitution rule:

$$\Gamma[h \mapsto H](x) = \begin{cases} H & \text{if } x = h \\ \Gamma(x) & \text{otherwise} \end{cases}$$

Pointer variables are considered to be a set of constant symbols and are thus given a fixed interpretation. The only thing that matters is that their interpretation is pairwise different. We assume that the pointer variables include a special name **null**. The set of pointer variables is denoted by the symbol P.

We will consider the semantics of propositional logic to be standard and the semantics of $\mathcal{T}_{\mathcal{B}}$ given, and thus just define the semantics of heap functions. To do this, we will first define the class of objects that will be used to interpret heap variables.

Definition 62 (Heap)**.** A heap over pointer variables P is a pair $H = \langle L, G \rangle$. G is a finite graph with vertices $V(G)$ and edges $E(G)$. $L : P \to V(G)$ is a labelling function mapping each pointer variable to a vertex of G. We define the cardinality of a heap to be the cardinality of the vertices of the underlying graph: $|H| = |V(G)|$.

Definition 63 (Singly Linked Heap)**.** A heap $H = \langle L, G \rangle$ is a singly linked heap iff each vertex has outdegree 1, except for a single sink vertex that has outdegree 0 and is labelled by **null**:

$$\begin{aligned} \forall v \in V(G).(\text{outdegree}(v) = 1 \land L(\textbf{null}) \neq v) \lor \\ (\text{outdegree}(v) = 0 \land L(\textbf{null}) = v) \end{aligned}$$

Having defined our domain of discourse, we are now in a position to define the semantics of the various heap functions introduced in Section 8.3.1. We begin with the functions examining the state of the heap and will use a standard structural recursion to give the semantics of the functions with respect to an implicit interpretation Γ, so that $[\![h]\!]\Gamma = \Gamma(h)$. We will use the shorthand $u \overset{n}{\to} v$ to say that if we start at node u, then follow n edges, we arrive at v. We also use $L(H)$ to select the labelling function

L from H:

$$\begin{aligned}
u \xrightarrow{n} v &\overset{\text{def}}{=} \langle u, v\rangle \in E^n \\
u \rightarrow^* v &\overset{\text{def}}{=} \exists n \geq 0 . u \xrightarrow{n} v \\
u \rightarrow^+ v &\overset{\text{def}}{=} \exists n > 0 . u \xrightarrow{n} v
\end{aligned}$$

Note that $u \xrightarrow{0} u$. The semantics of the heap functions are then:

$$\begin{aligned}
[\![\mathit{pathLength}(h, x, y)]\!]\Gamma &\overset{\text{def}}{=} \min\Big(\{n \mid L([\![h]\!]\Gamma)(x) \xrightarrow{n} L([\![h]\!]\Gamma)(y)\} \cup \{\infty\}\Big) \\
[\![\mathit{circular}(h, x)]\!]\Gamma &\overset{\text{def}}{=} \exists v \in V([\![h]\!]\Gamma) . L([\![h]\!]\Gamma)(x) \rightarrow^+ v \wedge v \rightarrow^+ L([\![h]\!]\Gamma)(x) \\
[\![\mathit{alias}(h, x, y)]\!]\Gamma &\overset{\text{def}}{=} [\![\mathit{pathLength}(h, x, y)]\!]\Gamma == 0 \\
[\![\mathit{isPath}(h, x, y)]\!]\Gamma &\overset{\text{def}}{=} [\![\mathit{pathLength}(h, x, y)]\!]\Gamma \neq \infty \\
[\![\mathit{isNull}(h, x)]\!]\Gamma &\overset{\text{def}}{=} [\![\mathit{pathLength}(h, x, \mathit{null})]\!]\Gamma == 0
\end{aligned}$$

Note that since the graph underlying H has outdegree 1, *pathLength* and *circular* can be computed in $O(|H|)$ time, or equivalently they can be encoded with $O(|H|)$ arithmetic constraints.

To define the semantics of the mutation operations, we will consider separately the effect of each mutation on each component of the heap – the labelling function L, the vertex set V and the edge set E. Where a mutation's effect on some heap component is not explicitly stated, the effect is id. For example, *assign* does not modify the vertex set, and so $\mathit{assign}_V = \mathsf{id}$. In the following definitions, we will say that $\mathrm{succ}(v)$ is the unique vertex such that $(v, \mathrm{succ}(v)) \in E(H)$.

$$\begin{aligned}
[\![\mathit{new}_V(h, x)]\!]\Gamma &\overset{\text{def}}{=} V([\![h]\!]\Gamma) \cup \{q\} \quad \text{where } q \text{ is a fresh vertex} \\
[\![\mathit{new}_E(h, x)]\!]\Gamma &\overset{\text{def}}{=} E([\![h]\!]\Gamma) \cup \{(q, \mathit{null})\} \\
[\![\mathit{new}_L(h, x)]\!]\Gamma &\overset{\text{def}}{=} L([\![h]\!]\Gamma)[x \mapsto q] \\
[\![\mathit{assign}_L(h, x, y)]\!]\Gamma &\overset{\text{def}}{=} L([\![h]\!]\Gamma)[x \mapsto L([\![h]\!]\Gamma)(y)] \\
[\![\mathit{lookup}_L(h, x, y)]\!]\Gamma &\overset{\text{def}}{=} L([\![h]\!]\Gamma)[x \mapsto \mathrm{succ}(L([\![h]\!]\Gamma)(y))] \\
[\![\mathit{update}_E(h, x, y)]\!]\Gamma &\overset{\text{def}}{=} (E([\![h]\!]\Gamma) \setminus \{(L([\![h]\!]\Gamma)(x), \mathrm{succ}(L([\![h]\!]\Gamma)(x)))\}) \cup \\
&\qquad \{(L([\![h]\!]\Gamma)(x), L([\![h]\!]\Gamma)(y))\}
\end{aligned}$$

8.4 Deciding Validity of SLH

We will now turn to the question of deciding the validity of an SLH formula, that is for some formula ϕ we wish to determine whether ϕ is a tautology or if there is some Γ such that $\Gamma \models \neg\phi$. To do this, we will show that SLH enjoys a finite model property and that the existence of a fixed-size model can be encoded directly as an arithmetic constraint.

Our high-level strategy for this proof will be to define progressively coarser equivalence relations on SLH heaps that respect the transformers and observation functions. The idea is that all of the heaps in a particular equivalence class will be equivalent in terms of the SLH formulae they satisfy. We will eventually arrive at an equivalence relation (homeomorphism) that is sound in the above sense and which is also guaranteed to have a small heap in each equivalence class.

From here on we will slightly generalise the definition of a singly linked heap and say that the underlying graph is weighted with weight function $W : E(H) \to \mathbb{N}$. When we omit the weight of an edge (as we have in all heaps until now), it is to be understood that the edge's weight is 1.

8.4.1 Sound Equivalence Relations

We will say that an equivalence relation $\approx$ is *sound* if the following conditions hold for each pair of pointer variables x, y and transformer τ:

$$\forall H, H' \cdot H \approx H' \Rightarrow \mathit{pathLength}(H, x, y) = \mathit{pathLength}(H', x, y) \land \tag{8.1}$$

$$\mathit{circular}(H, x) = \mathit{circular}(H', x) \land \tag{8.2}$$

$$\tau(H) \approx \tau(H') \tag{8.3}$$

The first two conditions say that if two heaps are in the same equivalence class, there is no observation that can distinguish them. The third condition says that the equivalence relation is inductive with respect to the transformers. There is therefore no sequence of transformers and observations that can distinguish two heaps in the same equivalence class.

We begin by defining two sound equivalence relations:

Definition 64 (Reachable Sub-Heap)**.** The reachable sub-heap $H|_P$ of a heap H is H with vertices restricted to those reachable from the pointer variables P:

$$V(H|_P) = \{v \mid \exists p \in P. \langle L(p), v\rangle \in E^*\}$$

Then the relation $\{\langle H, H'\rangle \mid H|_P = H'|_P\}$ is sound.

Definition 65 (Heap Isomorphism)**.** Two heaps $H = \langle L, G\rangle, H' = \langle L', G'\rangle$ are isomorphic (written $H \simeq H'$) iff there exists a graph isomorphism $f : G|_P \to G'|_P$ that respects the labelling function, i.e., $\forall p \in P. f(L(p)) = L'(p)$.

Example 8. H and H' are not isomorphic, even though their underlying graphs are.

$H:$

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet \\
x \arrow[u, dashed] & \mathbf{null} \arrow[u, dashed]
\end{tikzcd}$$

$H':$

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet \\
x \arrow[ur, dashed] & \mathbf{null} \arrow[u, dashed]
\end{tikzcd}$$

Theorem 66. *Heap isomorphism is a sound equivalence relation.*

8.4.2 Heap Homeomorphism

The final notion of equivalence we will describe is the weakest. Loosely, we would like to say that two heaps are equivalent if they are "the same shape" and if the shortest distance between pointer variables is the same. To formalise this relationship, we will be using an analogue of topological homeomorphism.

Definition 67 (Edge Subdivision)**.** A graph G' is a subdivison of G iff G' can be obtained by repeatedly subdividing edges in G, i.e., for some edge $(u, v) \in E(G)$ introducing a fresh vertex q and replacing the edge (u, v) with edges $(u, q), (q, v)$ such that $W'(u, q) + W'(q, v) = W(u, v)$. Subdivision for heaps is defined in terms of their underlying graphs.

We define a function *subdivide*, which subdivides an edge in a heap. As usual, the function is defined componentwise on the heap:

$$\begin{aligned}
\mathit{subdivide}_V(H, u, v, k) &= V \cup \{q\} \\
\mathit{subdivide}_E(H, u, v, k) &= (E \setminus \{(u, v)\}) \cup \{(u, q), (q, v)\} \\
\mathit{subdivide}_W(H, u, v, k) &= W(H)[(u, v) \mapsto \infty, (u, q) \mapsto k, (q, v) \mapsto W(H)(u, v) - k]
\end{aligned}$$

Definition 68 (Edge Smoothing)**.** The inverse of edge subdivision is called edge *smoothing*. If G' can be obtained by subdividing edges in G, then we say that G is a smoothing of G'.

Basically, edge *smoothing* is the dual of edge subdivision – if we have two edges $u \xrightarrow{n} q \xrightarrow{m} v$, where q is unlabelled and has no other incoming edges, we can remove q and add the single edge $u \xrightarrow{n+m} v$.

Example 9. H' is a subdivision of H.

$H:$

$$\begin{tikzcd}
\bullet \arrow[r, "3"] & \bullet \\
x \arrow[u, dashed] & \mathbf{null} \arrow[u, dashed]
\end{tikzcd}$$

$H':$

$$\begin{tikzcd}
\bullet \arrow[r, "1"] & \bullet \arrow[r, "2"] & \bullet \\
x \arrow[u, dashed] & & \mathbf{null} \arrow[u, dashed]
\end{tikzcd}$$

Lemma 69 (Subdividing an Edge Preserves Observations)**.** *If H' is obtained from H by subdividing one edge, then for any x, y we have:*

$$pathLength(H, x, y) = pathLength(H', x, y) \tag{8.4}$$

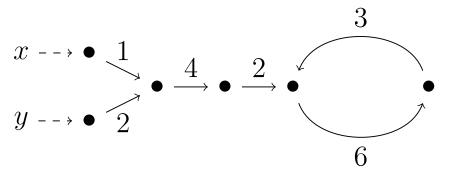
$$circular(H, x) = circular(H', x) \tag{8.5}$$

Definition 70 (Heap Homeomorphism)**.** Two heaps H, H' are homeomorphic (written $H \sim H'$) iff there there is a heap isomorphism from some subdivision of H to some subdivision of H'.

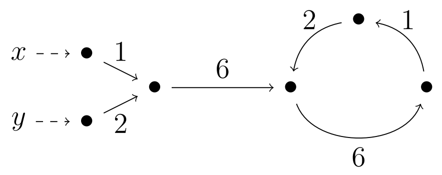
Intuitively, homeomorphisms preserve the topology of heaps: if two heaps are homeomorphic, then they have the same number of loops and the same number of "joins" (vertices with indegree ≥ 2).

Example 10. H and H' are homeomorphic, since they can each be subdivided to produce S.

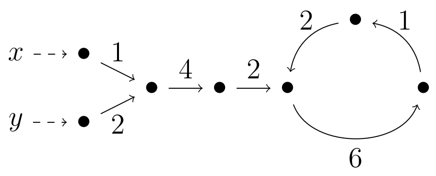
H :



H' :



S :



Lemma 71 (Transformers Respect Homeomorphism)**.** *For any heap transformer τ, if $H_1 \sim H_2$ then $\tau(H_1) \sim \tau(H_2)$.*

Proof. It suffices to show that for any transformer τ and single-edge subdivision s, the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "\tau"] \arrow[d, "s"'] & C \arrow[d, "s"] \\
B \arrow[r, "\tau"'] & D
\end{tikzcd}$$

We will check that $\tau \circ s = s \circ \tau$ by considering the components of each arrow separately and using the semantics defined in Section 8.3.3. The only difficult case is for *lookup*, for which we provide the proof in full. This case is illustrative of the style of reasoning used for the proofs of the other transformers.

$\tau = \mathit{lookup}(h, x, y)$**:** Now that we have weighted heaps, there are two cases for *lookup*: if the edge leaving $L(y)$ does not have weight 1, we need to first subdivide so that it does; otherwise the transformer is exactly as in the unweighted case, which can be seen easily to commute.

In the second (unweighted) case, all of the components commute due to id. Otherwise, *lookup* is a composition of some subdivision s' and then unweighted lookup: $\mathit{lookup} = \mathit{lookup}_U \circ s'$.

$$\begin{tikzcd}
A \arrow[d, "s'"'] \arrow[dr, "\mathit{lookup}"] & \\
B \arrow[r, "\mathit{lookup}_U"'] & C
\end{tikzcd}$$

Our commutativity condition is then:

$$(\mathit{lookup}_U \circ s') \circ s = s \circ (\mathit{lookup}_U \circ s')$$

We know that unweighted *lookup* commutes with arbitrary subdivisions, so

$$\begin{aligned}(\mathit{lookup}_U \circ s') \circ s &= s \circ (s' \circ \mathit{lookup}_U)\\ \mathit{lookup}_U \circ (s' \circ s) &= (s \circ s') \circ \mathit{lookup}_U\end{aligned}$$

But the composition of two subdivisions is a subdivision, so we are done.

□

Theorem 72. *Homeomorphism is a sound equivalence relation.*

Proof. This is a direct consequence of Lemma 69 and Lemma 71. □

8.4.3 Small Model Property

We would now like to show that for each equivalence class induced by $\sim$, there is a unique minimal element. We call that element the *kernel*.

Definition 73 (Kernel)**.** A kernel is a heap $H = (L, G)$ such that all the vertices in G are either labelled by L, or have at least two incoming edges.

In other words, a kernel is the maximally smoothed heap.

Theorem 74 (The Kernel is Unique)**.** *Each equivalence class induced by $\sim$ has a unique kernel.*

Proof. We can prove this by contradiction. Let's assume there are two such kernels K_1 and K_2 in an equivalence class. Then $K_1 \sim K_2$, and according to the homeomorphism definition, one is a subdivision of the other. Let's say K_1 is a subdivision of K_2. However, subdividing an edge introduces anonymous vertices with only one incoming edge. Thus K_1 is not a kernel. $\square$

As an alternative intuition for this, readers familiar with category theory can consider the category **SLH** of singly linked heaps, with edge subdivisions as arrows. The category **SLH** are singly linked heaps, and there is an arrow from one heap to another if the first can be subdivided into the second. To illustrate, Example 10 is represented in **SLH** by the following diagram:

$$\begin{tikzcd}
 & H \arrow[d] \\
H' \arrow[r] & S
\end{tikzcd}$$

Now for every pair of homeomorphic heaps $H_1 \sim H_2$ we know that there is some X that is a subdivision of both H_1 and H_2. Clearly if we continue subdividing edges, we will eventually arrive at a heap where every edge has weight 1, at which point we will be unable to subdivide any further. Let us call this maximally subdivided heap the *shell*, which we will denote by $\mathrm{Sh}(H_1)$. Then $\mathrm{Sh}(H_1) = \mathrm{Sh}(H_2)$ is the pushout of the previous diagram. Dually, there is some Y that both H_1 and H_2 are subdivisions of, and the previous diagram has a pullback, which we shall call the *kernel*. This is the heap in which all edges have been smoothed. The following diagram commutes, and since a composition of subdivisions and smoothings is a homeomorphism, all of the arrows (and their inverses) in this diagram are homeomorphisms. In fact, the $H_1, H_2, X, Y, \mathrm{Sh}$ and Ke are exactly an equivalence class:

$$\begin{tikzcd}
\mathrm{Ke} \arrow[dr] \arrow[drr, bend left] \arrow[ddr, bend right] & & & \\
 & Y \arrow[r] \arrow[d] & H_1 \arrow[d] \arrow[ddr, bend left] & \\
 & H_2 \arrow[r] \arrow[ur, dashed, leftrightarrow, "\sim"] \arrow[drr, bend right] & X \arrow[dr] & \\
 & & & \mathrm{Sh}
\end{tikzcd}$$

Lemma 75 (Kernels are Small)**.** *For any H, $|\mathrm{Ke}(H)| \leq 2 \times |P|$.*

Proof. Since Ke(H) is maximally smoothed, every unlabelled vertex has indegree ≥ 2. We will partition the vertices of H into named and unlabelled vertices:

$$\begin{aligned}
N =& \{v \in V(H) \mid \exists p \in P. L(p) = v\} \\
U =& \{u \in V(H) \mid \forall p \in P. L(p) \neq u\} \\
V(H) =& N \cup U
\end{aligned}$$

Then let $n = |N|$ and $u = |U|$. Now, the total indegree of the underlying graph must be equal to the total outdegree, so:

$$\begin{aligned}
\sum_{v \in V(H)} \mathrm{out}(v) &= \sum_{v \in V(H)} \mathrm{in}(v) \\
n + u &= \sum_{n \in N} \mathrm{in}(n) + \sum_{u \in U} \mathrm{in}(u) \\
&= \sum_{n \in N} \mathrm{in}(n) + 2u + k
\end{aligned}$$

where $k \geq 0$, since $\mathrm{in}(u) \geq 2$ for each u.

$$\begin{aligned}
n &= u + \underbrace{\sum_{n \in N} \mathrm{in}(n) + k}_{\geq 0} \\
n &\geq u
\end{aligned}$$

So $u \leq n \leq |P|$, hence $|\mathrm{Ke}(H)| = n + u \leq 2 \times |P|$. $\square$

Theorem 76 (SLH has Small Model)**.** *For any SLH formula $\forall h.\phi$, if there is a counterexample $\Gamma \models \neg\phi$, then there is $\Gamma' \models \neg\phi$ with every heap-sorted variable in Γ being interpreted by a homeomorphism kernel.*

Proof. This follows from Theorem 72 and Lemma 75. $\square$

We can encode the existence of a small model with an arithmetic constraint whose size is linear in the size of the SLH formula, since each of the transformers can be encoded with a constant sized constraint and the observation functions can be encoded with a constraint of size $O(|H|) = O(|P|)$. An example implementation of the constraints used to encode each atom is given in Section 8.6. We need one constraint for each of the theory atoms, which gives us $O(|P| \times |\phi|)$ constraints in total.

Corollary 77 (Decidability of SLH)**.** *If the background theory $\mathcal{T}_{\mathcal{B}}$ is decidable, then SLH is decidable.*

Proof. The existence of a small model can be encoded with a linear number of arithmetic constraints in $\mathcal{T}_{\mathcal{B}}$. □

8.5 Using SLH for Verification

Our intention is to use SLH for reasoning about the safety and termination of programs with potentially cyclic singly-linked lists:

- For safety, we annotate loops with safety invariants and generate VCs checking that each loop annotation is genuinely a safety invariant, i.e. (1) it is satisfied by each state reachable on entry to the loop, (2) it is inductive with respect to the program's transition relation, and (3) excludes any states where an assertion violation takes place (the assertions include those ensuring memory safety). The existence of a safety invariant corresponds to the notion of partial correctness: no assertion fails, but the program may never stop running.
- For termination, we provide ranking functions for each loop and generate VCs to check that the loops do terminate, i.e. the ranking function is monotonically decreasing with respect to the loop's body and (2) it is bounded from below. By combining these VCs with those generated for safety, we create a total-correctness specification.

The two additional items we must provide in order to be able to generate these VCs are a programming language and the strongest post-condition for formulae in SLH with respect to statements in the programming language. We do so next.

8.5.1 Programming Language

We use the sequential programming language in Fig. 8.3. It allows heap allocation and mutation, with v denoting a variable and *next* a pointer field. To simplify the presentation, we assume each data structure has only one pointer field, *next*, and allow only one-level field access, denoted by $v{\rightarrow}next$. Chained dereferences of the form $v{\rightarrow}next{\rightarrow}next\ldots$ are handled by introducing auxiliary variables. The statement *assert*(ϕ) checks whether ϕ (expressed in the heap theory described in Section 8.3) holds for the current program state, whereas *assume*(ϕ) constrains the program state.

$$\boxed{\begin{array}{lcl} \textit{datat} & := & \textit{struct } C\ \{(\textit{typ } v)^*\} \\ e & := & v \mid v{\rightarrow}\textit{next} \mid \text{new}(C) \mid \texttt{null} \\ S & := & v{=}e \mid v_1{\rightarrow}\textit{next}{=}v_2 \mid S_1; S_2 \mid \textit{if } (B)\ S_1 \textit{ else } S_2 \mid \\ & & \textit{while } (B)\ S \mid \text{assert}(\phi) \mid \text{assume}(\phi) \end{array}}$$

Figure 8.3: Programming Language

For convenience when using SLH in the context of safety and termination verification, the SLH functions we expose in the specification language are side-effect free. That is to say, we don't require the explicit heap h to be mentioned in the specifications.

8.5.2 Strongest Post-Condition

To create a verification condition from a specification, we first decompose the specification into Hoare triples and then compute the strongest post-condition to generate a VC in the SLH theory. Since SLH includes primitive operations for heap manipulation, our strongest post-condition is easy to compute:

$$\begin{aligned} \mathsf{SP}(x = y, \phi) &\stackrel{\text{def}}{=} \phi[h'/h] \wedge h = \textit{assign}(h', x, y) \\ \mathsf{SP}(x = y{\rightarrow}\textit{next}, \phi) &\stackrel{\text{def}}{=} \phi[h'/h] \wedge h = \textit{lookup}(h', x, y) \\ \mathsf{SP}(x = \text{new}(C), \phi) &\stackrel{\text{def}}{=} \phi[h'/h] \wedge h = \textit{new}(h', x, y) \\ \mathsf{SP}(x{\rightarrow}\textit{next} = y, \phi) &\stackrel{\text{def}}{=} \phi[h'/h] \wedge h = \textit{update}(h', x, y) \end{aligned}$$

In the definitions above, h' is a fresh heap variable. The remaining cases for SP are standard.

8.5.3 VC Generation Example

Consider the program in Figure 8.4, which has been annotated with a loop invariant. In order to verify the partial-correctness condition that the assertion cannot fail, we must check the following Hoare triples:

$$\{\top\}\ \text{x} = \text{y}\ \{\textit{isPath}(y, x)\} \tag{8.6}$$

$$\{\textit{isPath}(y, x) \wedge \neg\textit{isNull}(x)\}\ \text{x} = \text{x} \rightarrow \text{next}\ \{\textit{isPath}(y, x)\} \tag{8.7}$$

$$\{\textit{isPath}(y, x) \wedge \textit{isNull}(x)\}\ \textbf{skip}\ \{\textit{isPath}(y, x)\} \tag{8.8}$$

```
x = y;

while (x ≠ null) {
  {isPath(y, x)}
  x = x→next;
}

assert(isPath(y, x));
```

Figure 8.4: An annotated program.

Taking strongest post-condition across each of these triples generates the following SLH VCs:

$$\forall h.h' = \mathit{assign}(h, x, y) \Rightarrow \mathit{isPath}(h', y, x) \tag{8.9}$$

$$\forall h.\mathit{isPath}(h, y, x) \wedge \neg \mathit{isNull}(x) \wedge h' = \mathit{lookup}(h, x, x) \Rightarrow \mathit{isPath}(h', y, x) \tag{8.10}$$

$$\forall h.\mathit{isPath}(h, y, x) \wedge \mathit{isNull}(x) \Rightarrow \mathit{isPath}(h, y, x) \tag{8.11}$$

8.6 Implementation

For our implementation, we instantiate SLH with the theory of bit-vector arithmetic. Thus, according to Corollary 77, the resulting theory SLH[$\mathcal{T}_{\mathcal{BV}}$] is decidable. In this section, we provide details about the implementation of the decision procedure via a reduction to SAT.

To check validity of an SLH[$\mathcal{T}_{\mathcal{BV}}$] formula ϕ, we search for a small counterexample heap H. By Theorem 76, if no such small H exists, there is no counterexample and so ϕ is a tautology. We encode the existence of a small counterexample by constructing a SAT formula.

To generate the SAT formula, we instantiate every occurrence of the SLH[$\mathcal{T}_{\mathcal{BV}}$] functions with the functions shown in Figure 8.5. The structure that the functions operate over is the following, where N is the number of vertices in the structure and P is the number of program variables:

```
typedef int node;
typedef int ptr;

struct heap {
  ptr: node[P];
  succ : (node × int)[N];
  num_nodes: int;
}
```

The heap contains N nodes, of which num_nodes are allocated. Pointer variables are represented as integers in the range $[0, P-1]$ where by convention **null** $= 0$. Each pointer variable is mapped to an allocated node by the ptr array, with the restriction that **null** maps to node 0. The edges in the graph are encoded in the succ array where h.succ[n] = (m, w) iff the edge (n, m) with weight w is in the graph. For a heap with N nodes, this structure requires $3N + 1$ integers to encode.

The implementations of the SLH[$\mathcal{T_{BV}}$] functions described in Section 8.3.1 are given in Figure 8.5. Note that only `Alloc` and `Lookup` can allocate new nodes. Therefore if we are searching for a counterexample heap with at most $2P$ nodes, and our formula contains k occurrences of `Alloc` and `Lookup`, the largest heap that can occur in the counterexample will contain no more than $2P + k$ nodes. We can therefore encode all of the heaps using $6P + 3k + 1$ integers each.

When constructing the SAT formula corresponding to the SLH[$\mathcal{T_{BV}}$] formula, each of the functions can be encoded (via symbolic execution) as a formula in the background theory $\mathcal{T_B V}$ of constant size, except for `PathLength` which contains a loop. This loop iterates $N = 2P + k$ times and so expands to a formula of size $O(P)$. If the SLH[$\mathcal{T_{BV}}$] formula contains x operations, the final SAT formula in $\mathcal{T_{BV}}$ is therefore of size $x \times P$. We use CBMC [29] to construct and solve the SAT formula.

One important optimisation when constructing the SAT formula involves a symmetry reduction on the counterexamples. Since our encoding assigns names to each of the vertices in the graph, we can have multiple representations for heaps that are isomorphic. To ensure that the SAT solver only considers a single counterexample from each homeomorphism class, we choose a canonical representative of each class and add a constraint that the counterexample we are looking for must be one of these canonical representatives. We define the canonical form of a heap such that the nodes are ordered topologically and so that the ordering is compatible with the ordering on the program variables. Note that this canonical form is described in terms of a breadth-first traversal of the graph, which eliminates cycles.

$$\forall p, p' \in P . p < p' \Rightarrow \forall n, n' . L(p) \to^* n \wedge L(p') \to^* n' \Rightarrow n \leq n'$$
$$\forall n, n' . n \to n' \Rightarrow n \leq n'$$

Where $n \to^* n'$ means n' is reachable from n.

```
function NewNode(heap h)
    n ← h.num_nodes
    h.num_nodes ← h.num_nodes + 1
    h.succ[n] ← (null, 1)
    return n

function Subdivide(heap h, node a)
    n ← NewNode(h)
    (b, w) ← h.succ[a]
    h.succ[a] ← (n, 1)
    h.succ[n] ← (b, w - 1)
    return n

function Update(heap h, ptr x, ptr y)
    n ← h.ptr[x]
    m ← h.ptr[y]
    h.succ[n] ← (m, 1)

function Assign(heap h, ptr x, ptr y)
    h.ptr[x] ← h.ptr[y]

function Lookup(heap h, ptr x, ptr y)
    n ← h.ptr[y]
    (n', w) ← h.succ[n]
    if w ≠ 1 then
        n' ← Subdivide(h, n)
    h.ptr[x] ← n'

function Alloc(heap h, ptr x)
    n ← NewNode(h)
    h.ptr[x] ← n

function PathLength(heap h, ptr x, ptr y)
    n ← h.ptr[x]
    m ← h.ptr[y]
    distance ← 0
    for i ← 0 to h.num_nodes do
        if n = m then
            return distance
        else
            (n, w) ← h.succ[n]
            distance ← distance + w
    return ∞

function Circular(heap h, ptr x)
    n ← h.ptr[x]
    m ← h.succ[n]
    distance ← 0
    for i ← 0 to h.num_nodes do
        if m = n then
            return True
        else
            if n = null then
                return False
        m ← h.succ[m]
    return False
```

Figure 8.5: Implementation of the SLH[$\mathcal{T}_{\mathcal{BV}}$] functions

8.7 Motivation Revisited

In this section, we get back to the motivational examples in Figure 8.1 and express their safety invariants and termination arguments in SLH. As mentioned in Section 8.5.1, for ease of use, we don't mention the explicit heap h in the specifications.

In Figure 8.1a, assuming that the call to the *length* function ensures the state before the loop to be $\mathit{pathLength}(h, x, \texttt{null}) = n$, then a possible safety invariant is $\mathit{pathLength}(h, y, \texttt{null}) = n - i$. Note that this invariant covers both the case where the list pointed by x is acyclic and the case where it contains a cycle. In the latter scenario, given that $\infty - i = \infty$, the invariant is equivalent to $\mathit{pathLength}(h, y, \texttt{null}) = \infty$. A ranking function for this program is $R(i) = -i$.

The program in Figure 8.1b is safe with a possible safety invariant:

$$\mathit{pathLength}(h, z, \texttt{null}) == \mathit{pathLength}(h, t, \texttt{null}).$$

Similar to the previous case, this invariant covers the scenario where the lists pointed by x and y are acyclic, as well as the one where they are cyclic. In the latter situation, the program does not terminate.

For the example in Figure 8.1c, the *divides* function is safe and a safety invariant is:

$$\mathit{isPath}(x, \mathit{null}) \wedge \mathit{isPath}(z, \mathit{null}) \wedge \mathit{isPath}(y, \mathit{null}) \wedge \mathit{isPath}(y, z) \wedge \mathit{isPath}(x, w) \wedge$$
$$\neg \mathit{isNull}(y) \wedge (\mathit{pathLength}(x, w) + \mathit{pathLength}(z, \mathit{null}))\%\mathit{pathLength}(y, \mathit{null}) == 0.$$

Additionally, the function terminates as witnessed by the ranking function $R(w) = \mathit{pathLength}(w, \mathit{null})$.

Function *isCircular* in Figure 8.1c is safe and terminating with the safety invariant: $\mathit{pathLength}(l, p) \wedge \mathit{pathLength}(p, q) \wedge \mathit{isPath}(q, p) {\neq} \mathit{isPath}(l, \texttt{null})$, and lexicographic ranking function: $R(q, p) = (\mathit{pathLength}(q, \mathit{null}), \mathit{pathLength}(q, p))$.

8.8 Experiments

To evaluate the applicability of our theory, we created a tool for verifying that heaps don't lie: Shakira [92]. We ran Shakira on a collection of programs manipulating singly linked lists. This collections includes the standard operations of traversal, reversal, sorting etc. as well as the motivational examples from Section 8.2. Each of the programs in this collection is annotated with correctness assertions and loop invariants, as well as the standard memory-safety checks. One of the programs (the motivational program from Figure 8.1b) used a non-linear loop invariant, but this did not require any special treatment by Shakira.

To generate VCs for each program, we generated a Hoare proof and then used CBMC 4.9 [29] to compute the strongest post-conditions for each Hoare triple using symbolic execution. The resulting VCs were solved using Glucose 4.0 [3]. As well as correctness and memory safety, these VCs proved that each loop annotation was genuinely a loop invariant. For four of the programs, we annotated loops with ranking functions and generated VCs to check that the loops terminated, thereby creating a total-correctness specification.

None of the proofs in our collection relied on assumptions about the shape of the heap beyond that it consisted of singly linked lists. In particular, our safety proofs show that the safe programs are safe even in the presence of arbitrary cycles and sharing between pointers.

We ran our experiments on a 4-core 3.30 GHz Core i5 with 8 GB of RAM. The results of these experiments are given in Table 8.1.

The top half of the table gives the aggregate results for the benchmarks in which the specifications held, i.e., the VCs were unsatisfiable. These "safe" benchmarks

	LOC	#VCs	Symex(s)	SAT(s)	C/E
Safe benchmarks (UNSAT VCs)					
SLL (safe)	236	40	18.2	5.9	—
SLL (termination)	113	25	14.7	9.6	—
Counterexamples (SAT VCs)					
CLL (nonterm)	38	14	6.9	1.6	3
Null-deref	165	31	13.6	3.0	3
Assertion Failure	73	11	3.5	0.7	3.5
Inadequate Invariant	37	4	4.9	1.2	6

Table 8.1: Experimental results

Legend:

LOC	Total lines of code
#VCs	Number of VCs
Symex(s)	Total time spent in symbolic execution to generate VCs
SAT(s)	Total time spent in SAT solver
C/E	Average counterexample size (number of nodes)

are divided into two categories: partial- and total-correctness proofs. Note that the total-correctness proofs involve solving more complex VCs – the partial correctness proofs solved 40 VCs in 5.9 s, while the total correctness proofs solved only 25 VCs in 9.6 s. This is due to the presence of ranking functions in the total-correctness proofs, which by necessity introduces a higher level of arithmetic complexity.

The bottom half of the table contains the results for benchmarks in which the VCs were satisfiable. Since the VCs were generated from a Hoare proof, their satisfiability only tells us that the purported proof is not in fact a real proof of the program's correctness. However, Shakira outputs models when the VCs are satisfiable and these can be examined to diagnose the cause of the proof's failure. For our benchmarks, the counterexamples fell into four categories:

- Non-termination due to cyclic lists.
- Null dereferences.
- A correctness assertion (not a memory-safety assertion) failing.
- The loop invariant being inadequate, either by being too weak to prove the required properties, or failing to be inductive.

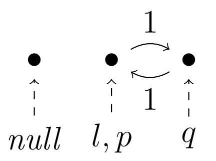
A counterexample generated by Shakira is given in Figure 8.6. This program is a variation on the motivational program from Figure 8.1c in which the programmer

```
int has_cycle(list l) {
  list p = l;
  list q = l→n;

  do {
    // Unwind loop to search
    // twice as fast!
    if (p != NULL) p = p→n;
    if (p != NULL) p = p→n;

    if (q != NULL) q = q→n;
    if (q != NULL) q = q→n;
    if (q != NULL) q = q→n;
    if (q != NULL) q = q→n;
  } while (p != q &&
           p != NULL &&
           q != NULL);

  return p == q;
}
```



Counterexample heap leading to non-termination.

Figure 8.6: A non-terminating program and the counterexample found by Shakira.

has tried to speed up the loop by unwinding it once. The result is that the program no longer terminates if the list contains a cycle whose size is exactly one, as shown in the counterexample found by Shakira.

These results show that discharging VCs written in SLH is practical with current technology. They further show that SLH is expressive enough to specify safety, termination and correctness properties for difficult programs. When the VCs require arithmetic to be done on list lengths, as is necessary when proving termination, the decision problem becomes noticeably more difficult. Our encoding is efficient enough that even when the VCs contain non-linear arithmetic on path lengths, they can be solved quickly by an off-the-shelf SAT solver.

8.9 Related Work

Research works on relating the shape of data structures to their numeric properties (e.g. length) follow several directions. For abstract interpretation based analyses, an abstract domain that captures both heap and size was proposed in [16]. The THOR tool [78, 79] implements a separation logic [88] based shape analysis and uses an off-the-shelf arithmetic analysis tool to add support for arithmetic reasoning. This approach is conceptually different from ours as it aims to separate the shape reasoning from the numeric reasoning by constructing a numeric program that explicitly tracks changes in data structure sizes. In [17], Boujjani et al. introduce the logic SLAD for reasoning about singly-linked lists and arrays with unbounded data, which allows to combine shape constraints, written in a fragment of separation logic, with data and size constraints. While SLAD is a powerful logic and has a decidable fragment, our main motivation for designing a new logic was its translation to SAT. A second motivation was the unrestricted sharing.

Other recent decidable logics for reasoning about linked lists were developed [17, 60, 77, 85, 100]. Piskac et al. provide a reduction of decidable separation logic fragments to a decidable first-order SMT theory [85]. A decision procedure for an alternation-free sub-fragment of first-order logic with transitive closure is described in [60]. Lahiri and Qadeer introduce the Logic of Interpreted Sets and Bounded Quantification (LISBQ) capable to express properties on the shape and data of composite data structures [72]. In [23], Brain et al. propose a decision procedure for reasoning about aliasing and reachability based on Abstract Conflict Driven Clause Learning (ACDCL) [41]. As they don't capture the lengths of lists, these logics are better suited for safety and less for termination proving.

In [9], Berdine et al. present a small model property for a fragment of separation logic with linked lists without explicit lengths. Their small model property says that it suffices to check if lists of lengths zero and two entail the formula (i.e. it suffices to unfold the list predicates 0 and 2 times). However if their fragment allowed imposing minimum lengths for lists, their small model result would be violated. In our case, since SLH allows adding explicit constraints on the lengths of lists (thus, one can impose minimum lengths), their small model property does not hold.

Chapter 9

Conclusions

We have shown that it is possible to automatically analyse a large class of C programs without generating false alarms. In particular, our analyses are able to soundly handle programs with very deep, complex loops whose behaviour depends on fixed-width integer semantics.

Our first analysis, acceleration, makes use of a sound under-approximation technique for loops in C programs with bit-vector semantics. The approach is very effective for finding deep counterexamples in programs that manipulate arrays, and compatible with a variety of existing verification techniques.

We extended this analysis with a technique that constrains the search space of an accelerated program, enabling BMC-based tools to prove safety using a small unwinding depth. To this end, we use trace automata to eliminate redundant execution traces resulting from under-approximating acceleration. Unlike other safety provers, our approach does not rely on over-approximation, nor does it require the explicit computation of a fixed point.

To build our second analysis, we developed a variety of methods based on second-order formulations of program properties. To do this, we defined the second-order SAT problem. We have shown that second-order SAT is a very expressive logic occupying a high complexity class. Despite its complexity, it can be reduced to the synthesis of finite-state programs, which allows us to exploit the observation that many formulae have simple satisfying assignments and that this corresponds to synthesising short programs. We have demonstrated that second-order SAT is well suited to program verification by directly encoding safety and liveness properties as second-order SAT formulae. We have also shown that other applications, such as superoptimisation and QBF solving, map naturally onto second-order SAT.

To solve these second-order formulae, we have presented a novel synthesis algorithm which uses a combination of symbolic model checking, explicit state model checking and stochastic search. Our experiments show that this combination is effective at finding short solutions to second-order SAT problems stemming from a range of problem domains.

Using this second-order SAT solver, we have shown how to precisely encode termination and safety analyses as second-order SAT problems. These encodings are bit-level accurate and never make false claims about a program's behaviour. As part of this encoding, we defined the notion of *danger invariants*, which allow us to find bugs without unrolling loops, and without introducing false alarms. Our experimental evaluation has shown that using second-order SAT to prove termination proving is tractable in practice.

In order to extend the reach of second-order SAT based analysers, we developed a theory of singly lists that can be decided with a SAT solver. To this end, we have presented the logic SLH for reasoning about potentially cyclic singly-linked lists. The main characteristics of SLH are the fact that it allows unrestricted sharing in the heap and can relate the structure of lists to their length, i.e. reachability constraints with numeric ones. As SLH is parametrised by the background arithmetic theory used to express the length of lists, we present its instantiation SLH[$\mathcal{T}_{\mathcal{BV}}$] with the theory of bit-vector arithmetic and provide a way of efficiently deciding its validity via a reduction to SAT. We empirically show that SLH is both efficient and expressive enough for reasoning about safety and (especially) termination of list programs. Since SLH's decision procedure is based on a reduction to SAT, it would be possible to use second-order SAT solving to automatically infer invariants and ranking functions for SLH programs.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17, 2013.

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. IJCAI'09, pages 399–404. Morgan Kaufmann, 2009.

[4] James Avery. Size-change termination and bound analysis. In *FLOPS*, pages 192–207, 2006.

[5] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *IFM*, volume 2999 of *LNCS*. Springer, 2004.

[6] Jason Baumgartner and Andreas Kuehlmann. Enhanced diameter bounding via structural transformations. In *Design, Automation and Test in Europe (DATE)*, pages 36–41. IEEE, 2004.

[7] Amir M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. *Logical Methods in Computer Science*, 6(3), 2010.

[8] Amir M. Ben-Amram and Samir Genaim. On the linear ranking problem for integer linear-constraint loops. In *POPL*, pages 51–62, 2013.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*, pages 97–109, 2004.

[10] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving existentially quantified horn clauses. In *CAV*, pages 869–882, 2013.

[11] Dirk Beyer. Status Report on Software Verification (Competition Summary SV-COMP 2014). In Erika Ábrahám and Klaus Havelund, editors, *TACAS*, volume 8413 of *LNCS*, pages 373–388. Springer, 2014.

[12] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309. ACM, 2007.

[13] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[14] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In *SAS*, pages 105–125, 2013.

[15] Bernard Boigelot. *Symbolic Methods for Exploring Infinite State Spaces.* PhD thesis, Université de Liège, 1999.

[16] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, pages 1–22, 2012.

[17] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *ATVA*, 2012.

[18] Marius Bozga, Radu Iosif, and Filip Konecný. Fast acceleration of ultimately periodic relations. In *CAV*, volume 6174 of *LNCS*, pages 227–242. Springer, 2010.

[19] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, 2005.

[20] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *ICALP*, pages 1349–1361, 2005.

[21] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In *VMCAI*, pages 113–129, 2005.

[22] Martin Brain, Tom Crick, Marina De Vos, and John Fitch. TOAST: Applying answer set programming to superoptimisation. In *ICLP*, pages 270–284, 2006.

[23] Martin Brain, Cristina David, Daniel Kroening, and Peter Schrammel. Model and proof generation for heap-manipulating programs. In *ESOP*, pages 432–452, 2014.

[24] M.F. Brameier and W. Banzhaf. *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer, 2007.

[25] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *CAV*, pages 413–429, 2013.

[26] Hong Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. Proving nontermination via safety. In *TACAS*, pages 156–171, 2014.

[27] Hong Yi Chen, Shaked Flur, and Supratik Mukhopadhyay. Termination proofs for linear simple loops. In *Static Analysis (SAS)*, pages 422–438. Springer, 2012.

[28] Alonzo Church. Logic, arithmetic, automata. In *Proc. Internat. Congr. Mathematicians*, pages 23–35. Inst. Mittag-Leffler, Djursholm, 1962.

[29] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176. Springer, 2004.

[30] Michael Codish and Samir Genaim. Proving termination one loop at a time. In *WLPE*, pages 48–59, 2003.

[31] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *TACAS*, pages 236–250, 2010.

[32] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.

[33] Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS*, pages 47–61, 2013.

[34] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[35] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96, 1978.

[36] C. David, D. Kroening, and M. Lewis. Propositional Reasoning about Safety and Termination of Heap-Manipulating Programs. In *ESOP*, 2015.

[37] C. David, D. Kroening, and M. Lewis. Unrestricted Termination and Non-Termination Arguments for Bit-Vector Programs. In *ESOP*, 2015.

[38] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[39] Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. A general framework for automatic termination analysis of logic programs. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):117–156, 2001.

[40] Edsger W. Dijkstra et al. From predicate transformers to predicates, April 1982. Tuesday Afternoon Club Manuscript EWD821.

[41] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract conflict driven learning. In *POPL*, pages 143–154, 2013.

[42] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *TCAD*, 27(7):1165–1178, July 2008.

[43] R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. Amer Mathematical Society, June 1974.

[44] Alain Finkel and Jrme Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FST-TCS 2002*, volume 2556 of *LNCS*, pages 145–156. Springer, 2002.

[45] E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2005. `www.qbflib.org`.

[46] Faustino Gomez and Risto Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:5–317, 1997.

[47] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.

[48] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385, 2009.

[49] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[50] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.

[51] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *POPL*, pages 147–158, 2008.

[52] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS*, volume 1503 of *LNCS*, pages 200–214. Springer, 1998.

[53] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. Alternation for termination. In *SAS*, pages 304–319, 2010.

[54] Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. Linear ranking for linear lasso programs. In *ATVA*, pages 365–380, 2013.

[55] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *SAS*, volume 5673 of *LNCS*, pages 69–85. Springer, 2009.

[56] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.

[57] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. *CoRR*, abs/1308.4767, 2013.

[58] Hossein Hojjat, Radu Iosif, Filip Konecny, Viktor Kuncak, and Philipp Ruemmer. Accelerating interpolants. In *ATVA*, volume 7561 of *LNCS*, pages 197–202, 2012.

[59] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *CSL*, pages 160–174, 2004.

[60] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *CAV*, pages 756–772, 2013.

[61] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[62] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.

[63] Ina Kraan, David Basin, and Alan Bundy. Logic program synthesis via proof planning. In *Logic Program Synthesis and Transformation*, pages 1–14. 1993.

[64] D. Kroening and M. Lewis. Second-Order SAT Solving using Program Synthesis. *ArXiv e-prints*, September 2014.

[65] D. Kroening, M. Lewis, and G. Weissenbacher. Proving Safety with Trace Automata and Bounded Model Checking. *ArXiv e-prints*, October 2014.

[66] Daniel Kroening. CBMC. `http://www.cprover.org/cbmc/`.

[67] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *CAV*, volume 8044 of *LNCS*, pages 381–396. Springer, 2013.

[68] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *CAV*, pages 89–103, 2010.

[69] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *VMCAI*, volume 2575 of *LNCS*, pages 298–309. Springer, 2003.

[70] Daniel Kroening and Georg Weissenbacher. Counterexamples with loops for predicate abstraction. In *CAV*, volume 4144 of *LNCS*, pages 152–165. Springer, 2006.

[71] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *ASE*, pages 389–392. ACM, 2007.

[72] Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL*, pages 171–182, 2008.

[73] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming.* Springer, 2002.

[74] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving termination of imperative programs using Max-SMT. In *FMCAD*, pages 218–225, 2013.

[75] Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. Termination analysis with algorithmic learning. In *CAV*, pages 88–104, 2012.

[76] Jan Leike and Matthias Heizmann. Ranking templates for linear loops. In *TACAS*, pages 172–186, 2014.

[77] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap structures and data. In *POPL*, pages 611–622, 2011.

[78] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. THOR: A tool for reasoning about shape and arithmetic. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 428–432, 2008.

[79] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, pages 211–222, 2010.

[80] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, March 1971.

[81] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[82] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[83] Greg Nelson. A generalization of Dijkstra's calculus. *TOPLAS*, 11(4):517–561, 1989.

[84] Aditya V. Nori and Rahul Sharma. Termination proofs from tests. In *ESEC/SIGSOFT FSE*, pages 246–256, 2013.

[85] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In *CAV*, pages 773–789, 2013.

[86] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[87] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS*, pages 32–41, 2004.

[88] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[89] Andrey Rybalchenko. ARMC. `http://www7.in.tum.de/~rybal/armc`.

[90] Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *SAS*, volume 6887 of *LNCS*, pages 233–248. Springer, 2011.

[91] Peter Schrammel, Tom Melham, and Daniel Kroening. Chaining test cases for reactive system testing. In *ICTSS*, pages 133–148, 2013.

[92] Shakira. Hips Don't Lie, 2006.

[93] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, pages 88–105, 2014.

[94] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[95] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[96] `http://sv-comp.sosy-lab.org/2015/`.

[97] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. 42:230–265, 1936.

[98] Caterina Urban. The abstract domain of segmented ranking functions. In *SAS*, pages 43–62. Springer, 2013.

[99] Kuat Yessenov, Ruzica Piskac, and Viktor Kuncak. Collections, cardinalities, and relations. In *Verification, Model Checking, and Abstract Interpretation*, pages 380–395. Springer, 2010.

[100] Greta Yorsh, Alexander Moshe Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. *J.Log.Alg.Prog.*, 73(1-2), 2007.