

# **SYMBOLIC EXECUTION FOR ADVANCED PROGRAM REASONING**

**VIJAYARAGHAVAN MURALI**

**NATIONAL UNIVERSITY OF SINGAPORE**

**2014**

**SYMBOLIC EXECUTION FOR ADVANCED  
PROGRAM REASONING**

**VIJAYARAGHAVAN MURALI**

*B.Comp. (Hons.), NUS, 2009*

**A THESIS SUBMITTED**

**FOR THE DEGREE OF DOCTOR OF  
PHILOSOPHY**

**DEPARTMENT OF COMPUTER SCIENCE**

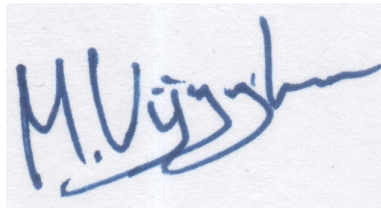
**NATIONAL UNIVERSITY OF SINGAPORE**

2014

# Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

A handwritten signature in blue ink, appearing to read 'M. Vijayaraghavan Murali', is centered on the page. The signature is written in a cursive style with a long horizontal stroke extending to the right.

---

Vijayaraghavan Murali

Monday 25<sup>th</sup> August, 2014

# Acknowledgment

First and foremost I would like to thank Professor Joxan Jaffar, who has been not only my advisor, but also a mentor and role-model. He has supported and motivated me throughout my Ph.D., during which I learned numerous things from him about research, teaching, career and life in general. I thank Baba (God) for bringing this great person into my life.

A special thanks goes to my old teammates Jorge Navas and Andrew Santosa who showed me that a good researcher needs to first be a good engineer – we together built the TRACER framework which helped actualise many ideas in this thesis.

I thank my other collaborators Satish Chandra, Duc-Hiep Chu, Nishant Sinha and Emina Torlak for showing me the breadth of research in this field and working together to solve many interesting problems.

I thank Professors Wei-Ngan Chin, Jin-Song Dong, Sanjay Jain, Siau-Cheng Khoo, Abhik Roychoudhury, Weng-Fai Wong, Roland Yap and many others for providing valuable insights through teaching. I also thank Professor Razvan Voicu for directing me to Joxan at the right time in life.

I thank Rasool Maghareh, Gregory Duck, Asankhaya Sharma, Pang Long, Marcel Böhme, Konstantin Rubinov and other colleagues and friends in the lab for their lively discussions. I thank my 6-year housemate Thyagu for his company and being almost a brother to me. I also thank anyone who would have helped me but I might have forgotten inadvertently.

Last but not least, I cannot find words to express thanks for my Amma (mother) and Appa (father) who gave all they had, and more, to see their son titled Ph.D. They are, simply put, my life and to whom I dedicate this thesis...

*To my parents Meera and Murali, with Baba's blessings*

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview of Current Techniques . . . . .	2
1.2	Overview of Symbolic Execution . . . . .	6
1.3	Thesis Contributions . . . . .	8
<b>2</b>	<b>Preliminaries</b>	<b>12</b>
2.1	Symbolic Execution . . . . .	12
2.2	Interpolation and Witnesses . . . . .	16
2.3	Implementation: TRACER . . . . .	22
<b>3</b>	<b>Backward Slicing</b>	<b>25</b>
	<b>Part I: Static Backward Slicing</b>	<b>26</b>
3.1	Motivating Example . . . . .	29
3.2	Background . . . . .	32
3.3	Algorithm . . . . .	33
3.4	Experimental Evaluation . . . . .	41
3.5	Related Work . . . . .	43
3.6	Summary . . . . .	45
	<b>Part II: Slice-based Program Transformation</b>	<b>46</b>
3.7	Related Work . . . . .	48
3.8	Basic Idea . . . . .	51
3.9	Background . . . . .	57
3.10	Algorithm . . . . .	57
3.11	Experimental Evaluation . . . . .	66

3.12	Summary . . . . .	70
<b>4</b>	<b>Concolic Testing</b>	<b>71</b>
4.1	Related Work . . . . .	73
4.2	Running Example . . . . .	75
4.3	Background . . . . .	79
4.4	Algorithm . . . . .	80
4.5	Experimental Evaluation . . . . .	86
4.6	Summary . . . . .	94
<b>5</b>	<b>Interpolation-based Verification</b>	<b>95</b>
5.1	Examples . . . . .	97
5.2	Background . . . . .	101
5.3	Algorithm . . . . .	102
5.4	Experimental Evaluation . . . . .	106
5.5	Related Work and Discussion . . . . .	110
5.6	Summary . . . . .	112
<b>6</b>	<b>Trace Understanding</b>	<b>113</b>
6.1	Related Work . . . . .	123
6.2	Background . . . . .	126
6.3	Algorithm . . . . .	127
6.4	Experimental Evaluation . . . . .	136
6.5	Summary . . . . .	143
<b>7</b>	<b>Conclusion</b>	<b>144</b>
7.1	Future Directions . . . . .	146

# Summary

This thesis aims to address a number of program reasoning problems faced every day by programmers, using the technique of *symbolic execution*. Symbolic execution is a method for program reasoning that executes the program with symbolic inputs rather than actual data. It has the advantage of avoiding “infeasible” paths in the program (i.e., paths that cannot be exercised for any input), exploring which could provide spurious information about the program and mislead the programmer. However, as symbolic execution considers the feasibility of individual paths, the number of which could be exponential in general, it suffers from *path explosion*. To tackle this, we make use of the technique of *interpolation*, which was recently developed to alleviate path explosion by intelligently pruning the exploration of certain paths.

In this thesis, we investigate the following problems, elaborate challenges that our method faces in solving each problem, and show with evidence how our method is either better than current state-of-the-art techniques or benefits them significantly:

- **Backward Slicing:** the (static) slice of a program with respect to a particular variable at a program point is, informally, the subset of program statements that might affect the value of said variable at that point. The challenge here is to find the right balance between precision of slicing information and efficiency. Addressing this challenge, we formulate the *most precise* slicing algorithm that works with reasonable efficiency. Inspired by this result, we extend our method to go beyond static slicing, by introducing the notion of “Tree slicing” that produces a more general *transformation* of the program compared to static slicing. We show how tree slicing can be much more powerful than static slicing in reducing the program’s search space.
- **Concolic Testing:** recently, a technique called *concolic testing* was proposed to automatically generate test cases that maximise coverage. Concolic testing also suffers from path explosion as it aims to test every path



in the program, which could be exponential in number. Employing interpolation in this setting fails to provide much benefit, if any, due to the poor formation of interpolants from test cases the concolic tester executes. Thus, we introduce a novel algorithm to *accelerate* the formation of interpolants which, for the first time, brings to concolic testing the exponential benefit that interpolation is known for.

- **Interpolation-based Verification:** verifying a program is the process of proving that a program satisfies a given property. Recently, symbolic execution has gained traction in verification due to its ability to avoid infeasible paths, exploring which may result in spurious “false-positives”. We conjecture that this aversion of infeasible paths hinders the discovery of good interpolants, which are vital in pruning the search space in future. We formulate a new strategy for symbolic execution that temporarily ignores the infeasibility of paths in pursuit of better interpolants. Although this may seem antithetical to the principle of symbolic execution, our results show this “lazy” method of symbolic execution that ignores infeasibilities is able to outperform the canonical method significantly. This unprecedented result opens up a new dimension for symbolic execution and interpolation based reasoning.
- **Trace Understanding:** understanding execution traces (typically error traces) has been a nightmare for programmers, mainly due to long loop iterations in the trace. We propose a new method to aid in the understanding of traces by compressing loops using *invariants* that preserve the semantics of the original trace with respect to a “target” (e.g., an assertion violated by an error trace). The novelty of this method is that if we are unable to find such an invariant, we dynamically *unroll* the loop and attempt the discovery at the next iteration, where we are more likely to succeed as the loop stabilises towards an invariant.

# List of Tables

3.1	Results on Intel 3.2Gz 2Gb. <sup>1</sup> timeout after 2 hours or 2.5 Gb of memory consumption . . . . .	42
3.2	Statistics about the PSS-CFG . . . . .	66
3.3	Experiments on the PSS-CFG for concolic testing . . . . .	68
3.4	Experiments on the PSS-CFG for verification . . . . .	69
5.1	Verification Statistics for Eager and Lazy SE (A T/O is 180s (3 mins)) . . . . .	108
6.1	Trace statistics for our experiments. %C: percentage compression, #U: number of unrolls until compression was achieved (inner loop unrolls, if any) . . . . .	136
6.2	Trend with varying loop bounds for cdaudio and floppy . . . . .	141

# List of Figures

2.1	(a) A program to swap two integers (b) Its transition system . . .	14
2.2	Symbolic Execution Tree of the program in Fig. 2.1 . . . . .	15
2.3	(a) A verification problem (b) Its full symbolic execution tree . .	20
2.4	Building the Symbolic Execution Tree with Interpolation (WP) .	21
2.5	Architecture of TRACER . . . . .	23
3.1	(a) A program and its transition system, (b) its naive sym- bolic execution tree (SET) for slicing criterion (underlined state- ments) $\langle \ell_9, \{z\} \rangle$ . . . . .	29
3.2	Interpolation-based Symbolic Execution Tree for Fig. 3.1 . . . .	30
3.3	Main Abstract Operations for $\mathcal{D}^\omega$ . . . . .	34
3.4	Path-Sensitive Backward Slicing Analysis . . . . .	37
3.5	A program and its symbolic execution tree . . . . .	51
3.6	The PSS-CFG and corresponding transformed program for Fig. 3.5 . . . . .	54
3.7	Symbolic execution interleaved with dependency computation to produce the SE tree . . . . .	59
3.8	Transformation rules to produce the final PSS-CFG . . . . .	63
4.1	A program and its symbolic execution tree . . . . .	76
4.2	A Generic Concolic Tester . . . . .	80
4.3	Symbolic execution with interpolation along a path . . . . .	81
4.4	A Generic Concolic Tester with Pruning . . . . .	83
4.5	Timing for (a) cdaudio (b) diskperf (c) floppy (d) kbfiltr. X-axis: Paths, Y-axis: time in seconds . . . . .	89

4.6	Subsumption for (a) cdaudio (b) diskperf (c) floppy (d) kbfiltr. X-axis: Paths, Y-axis: % subsumption . . . . .	91
4.7	Extra coverage provided for (a) cdaudio (b) diskperf (c) floppy (d) kbfiltr by our method. X-axis: Crest path coverage, Y-axis: Additional path coverage from subsumption. . . . .	93
5.1	Proving $y \leq n$ : Eager vs Lazy . . . . .	97
5.2	A Program and its (Eager) SE Tree with Learning . . . . .	99
5.3	Lazy SE Tree with Learning . . . . .	100
5.4	A Framework for Lazy Symbolic Execution with Speculative Abstraction . . . . .	103
6.1	Hoare triples generated for the program <code>for_bounded_loop1.c</code>	118
6.2	(a) Program with nested loops (b) Its compressed trace . . . . .	120
6.3	Loop Compression with Invariants . . . . .	128
6.4	Basic Individually Invariant Discovery . . . . .	131
6.5	Invariant Generalisation using Weakest Precondition . . . . .	133
6.6	Symbolic Execution trees for the invariants in Fig. 6.2(b) . . . . .	134
6.7	The SSH client program, the error trace and the compressed trace	137
6.8	The SSH server program, the error trace and the compressed trace	139

# Chapter 1

## Introduction

*“The most important property of a program is whether it accomplishes the intentions of its user”*, writes C.A.R. Hoare in his seminal article [55] laying the foundations of formal program reasoning. It is widely accepted that this property is the *“Holy Grail”* of modern computer science.

Today, every programmer endeavours to achieve this goal at every stage of software production. While developing the software, the programmer tries to make sure that bugs are not unwittingly introduced into the code, although this sentence is an utter understatement of the complexity of the problem. Once the software is developed, the programmer then tries to increase confidence of its correctness by designing test cases that effectively explore the code. In case a bug is found, the programmer has to typically reason about a particular “error trace” that failed to comply with his/her intentions for the software.

On this note, a recent study [4] by Cambridge University showed that “software developers spend 50% of their programming time finding and fixing bugs” and that “the global cost of debugging software has risen to \$312 billion annually”. Despite this, bugs still manifest regularly in software shipped today. For instance, the infamous “Heartbleed” bug [5], found just weeks before the time of writing of this thesis, was the result of the lack of a bounds check, which caused a read overflow and potentially leaked sensitive information to attackers. More serious bugs have resulted in the loss of huge amounts of money or worse, human life [3].

Thus, it is of utmost importance to develop techniques to reason about pro-

grams and expose these bugs before the software is made available for public use. In a broad sense, the whole area of program analysis was developed over the past few decades for this purpose. A comprehensive survey of the entire field would appear daunting at this point, as there have been hundreds, if not thousands, of papers and books contributing techniques such as, to list a few, software model checking and abstract interpretation [27, 37], program slicing [104, 72], automated testing and debugging [49, 97, 108] and more.

## 1.1 Overview of Current Techniques

The goal of this thesis is to contribute in the following areas of program reasoning: program slicing, testing, verification and trace understanding. We briefly survey some traditional and contemporary techniques in each area.

### Program Slicing

Slicing, as defined by Weiser [104], is a technique that identifies the parts of a program that potentially affect the values of specified variables at a specified program point—the *slicing criterion*. This is sometimes referred to as *backward* slicing. Since Weiser’s original definition, many variants of the notion of slicing have been proposed, with different methods to compute them (see [103] for a survey). An important distinction is that between *static* and *dynamic* slicing [72], where the former does not assume any input provided to the program, and the latter assumes a particular input. Our focus here is on static backward slicing, which was originally intended to help programmers in debugging.

Static slicing was initially performed in [104] using data-flow analysis, by computing consecutive sets of indirectly relevant statements according to data and control dependencies. A different method that applies reachability analysis on the Program Dependence Graph (PDG) was proposed in [86, 42]. The problem of *interprocedural* slicing was later addressed in [57, 58], which proposed the idea of using a System Dependence Graph (SDG). The key argument was that slices computed by previous works were too imprecise due to not being able to distinguish between a realisable and non-realisable calling context.

Parallel to this, the framework of *abstract interpretation* was developed by [27], which simulates execution of the program on an abstract domain that shares a Galois-connection with the concrete domain. Once a fixed point is reached in the abstract domain, several concrete states can be combined to a single abstract state and the process terminates. Slicing can be formulated in abstract interpretation by defining the abstract domain to be the set of all possible *dependency variables* at a program point.

Today, slicing is being applied in program testing, differencing, maintenance, debugging, optimisation etc. However, the main problem still being faced is that slices are bigger than expected and sometimes too big to be useful, as [10] experimentally found out. One of the most important reasons for imprecision is the lack of consideration of the feasibility of program paths, many of which could be *infeasible* (i.e., not executable for any input), similar to the claim laid by [57, 58] for calling contexts. Our work aims to address this issue.

## Program Testing

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results [54]. The process of testing executes a given program with some inputs, and the objective is to find bugs or validate the program *with respect to the given inputs*. Indeed, Dijkstra expressed in his notes [35] that “testing can only prove the presence of bugs, but not their absence”. Nevertheless, testing is the oldest and still the most commonly used method to ensure software quality. Traditionally, testing was carried out manually, by programmers writing test cases themselves based on their understanding of the code (of course, this practice exists even today).

Automated testing methods such as random testing [102, 48], also called “fuzzing”, were introduced to generate *random* inputs with an aim to make the program crash or observe for memory leaks. This has the advantage of not requiring the source code of the program (referred to as “black-box” testing), and hence can be readily applied to test large applications such as C compilers [107]. Although random testing has helped in detecting various bugs throughout history, the randomness of inputs used in fuzzing is often seen as a disadvantage,

as catching a boundary value condition with random inputs is highly unlikely. A primitive fuzzer may also have poor code coverage; for example, if the input to a program is a file and its checksum, and the fuzzer generates random files and random checksums, only the checksum validation code in the program will be tested.

More recently, a technique called Directed Automated Random Testing (DART) [49, 97] was proposed as an alternative to random testing. DART executes the program with a given input, and obtains a formula describing the program path that was executed by the input (making this a “white-box” testing technique). Then, it makes sure to not execute the same path by *negating* one of the branches in the path formula, solving the new formula using a theorem prover, and generating (random) test inputs that satisfy the new formula. This has been shown to significantly increase the coverage of random testing [16].

An important technical problem with DART, also referred to as “concolic testing”, or more formally “dynamic symbolic execution”, is that as it aims to test every path in the program, it can run into *path explosion*. In this thesis, we address this issue.

## Program Verification

Verification is the process of constructing a *mathematical proof* that a program satisfies a given property. Properties come in two types: safety (i.e., those that state that something “bad” will never happen) and liveness (i.e., those that state that something “good” will eventually happen). In this work, we are only concerned with safety properties.

The seminal work by Hoare [55] established the foundations of reasoning by which to prove a program correct. For each building block of the program, a *Hoare triple*—an assume-guarantee style proof—is computed, which can then be composed with other such triples to construct a proof for the program. The disadvantage of this method is that a significant amount of manual effort is needed in the form of invariants and user-assertions.

Model checking [37] was proposed as a technique to *automatically* verify hardware designs, by typically constructing a finite state machine of the hard-



ware model and reducing the problem to graph search. In the case of software systems, which are typically infinite-state, abstraction has to be employed to make the model finite. This can result in spurious counter-examples (i.e., false positives). Recent techniques such as *Counter-Example Guided Abstraction Refinement* (CEGAR) [24, 9] have addressed this by starting with a coarse model of the program and then refining the model by analysing the spurious counter-examples, until a “real”<sup>1</sup> counter-example is found.

Recently, the technique of *symbolic execution*, which we will use in this work, has gained momentum in program verification [59]. It presents a dual approach to CEGAR, by starting with the concrete model of the program and removing irrelevant facts from it that are not needed for the proof. The main advantage of symbolic execution is that it avoids the expensive computation of the abstract post operation as in CEGAR.

## Trace Understanding

When a program fails, the cause of the failure is often buried in a long, hard-to-understand error trace. For this reason, techniques to help with understanding an error trace are valuable in debugging. “Understanding” is a very subjective term, so we make it concrete: our focus is on the *loop iterations* that typically make the trace long, and our goal is to compress them. Since this is a relatively narrow area of research, prior work in this area is limited.

Traditional methods of trace compression include dynamic slicing [72] on the variables in the assertion violated by the error trace. Dynamic slicing, however, only removes statements that it can deem irrelevant through *dependency* information (data or control flow). It does not reason about the *semantics* of the trace. This weakness was addressed recently in [39], that computed so-called “error invariants”—abstractions of the state that are still sufficient to violate the assertion—at each point along the trace. If two points have the same error invariants, any intervening statement is deemed irrelevant to the error, as the reason the trace violated the assertion has not changed between the two points.

---

<sup>1</sup>We say “real” (in quotes) because undecidability restricts any software verification method from being complete; the method can sometimes fail to prove or disprove a property.

However, these error invariants are not guaranteed to be loop invariants. Thus, the practical problem of long loop iterations in error traces still remains directly unaddressed, which is the motivation for our work in this area.

## 1.2 Overview of Symbolic Execution

So far, we have just seen the “tip of the iceberg” in the area of program reasoning. Even at this level, many static analyses often suffer from the problem of imprecision, mainly due to the assumption that all program paths are executable. Many paths, in fact, are not executable (or *feasible*) for any input because of conflicts in the logic of statements along the path. Gathering analysis information from these paths gives rise to spurious results, which may mislead the programmer. The art of analysing programs paying heed to whether individual paths are feasible or not is commonly referred to as *path sensitivity*.

It is folklore that path sensitive analyses are much more precise than path insensitive analyses. Hence it is natural to wonder “why are not all analyses path sensitive?” The reason is that path sensitivity suffers from a major problem: the number of paths to explore in a program is in general exponential in the number of branches. Considering the feasibility of each path to derive analysis information results in an exponential blowup. This is referred to as the *path explosion* problem and it severely limits the scalability of path sensitive analyses.

Due to this, many program analyses are either *path insensitive* or use some heuristics to skirt the path explosion issue. For instance, many state-of-the-art slicers available today (e.g., [25]) are path insensitive, and concolic testers which are (or rather, must be) path sensitive (e.g., [16]) use metrics such as branch coverage (as opposed to path coverage) to measure the quality of their testing procedure. That is, they forfeit the goal of generating tests to exercise every program path and instead target the much easier goal of generating tests to just exercise every branch (i.e., basic block) in the program. Thus, *there is much need to perform path sensitive program analyses efficiently*.

In this thesis, we employ a technique called *symbolic execution* to address

these problems. Symbolic execution [71], as the name implies, executes a program not with actual inputs but with symbolic inputs. The program statements that are encountered during this execution are collected in a *first-order logic* (FOL) formula called the *path condition*<sup>2</sup>. This formula is the crux of the versatility of symbolic execution, as it can be analysed to derive a host of information. For instance, it can be checked for satisfiability in order to infer whether the corresponding path is *feasible* (i.e., the path can be exercised by some input), it can be checked for the existence of bugs (i.e., assertion violations), it can be used to compute variable dependencies along the path, compute the (abstract) execution time of the path, and so on. Such information can be collected across multiple symbolic paths to derive some property about a program point, variable or even the whole program.

The key advantage of symbolic execution is that it can avoid the exploration of paths that are not feasible by stopping and backtracking the moment unsatisfiability is detected in the path condition. This offers a natural way to perform precise path sensitive analysis. However it suffers from the path explosion problem, as it attempts to consider the feasibility of every single path in the program. To address this problem, we use the technique of *interpolation* [28, 81, 66], which has been recently employed to mitigate state space blowup in model checking, and the concept of *witnesses* [66].

Briefly, the high level idea of interpolation and witnesses is as follows<sup>3</sup>. Whenever symbolic execution explores an entire tree of path arising from a node, we “learn” some relevant information about the tree— for instance, in the case of testing, we learn the essence of why the tree is bug-free. This information, called the *interpolant*, can be learned from the path condition of the tree’s root node, and is typically much more succinct than (formally, an *abstraction* of) its path condition. We also compute what is called a *witness*, a formula which describes the (sub)-analysis of the tree.

Then, when reaching the same program point through a different path, we check if the stored interpolant and witness are implied by the new path condi-

---

<sup>2</sup>The variables in constraints arising from the statements are assumed to be implicitly existentially quantified.

<sup>3</sup>Interpolation and witnesses are explained in full detail in the rest of the thesis.

tion. If the implication holds, the node can be *subsumed* (or *covered*), because it can be guaranteed to produce the same analysis result if explored. This can result in exponential savings because the entire tree of paths arising from the program point is pruned due to subsumption. The key insight is that the subsumed node can *reuse* the analysis computed previously by the subsuming node. Interpolation is therefore critical to the scalability of symbolic execution.

If the subsumption test failed (i.e., the entailment does not hold), symbolic execution will naturally perform node splitting and duplicate all successors of the node until the next merge point.

### 1.3 Thesis Contributions

The thesis that is explored in this work is the following: *using symbolic execution with interpolation, we can develop efficient and powerful techniques for path sensitive analysis for a variety of program reasoning problems.*

In Chapter 2, we setup the formal background of symbolic execution and present our framework TRACER, first demonstrated in [64], on which the ideas in this thesis are implemented. In Chapter 3, we present our two main contributions to the area of slicing: first, a method to efficiently perform static path-sensitive backward slicing, published in [65], and second, a more powerful program transformation technique based on path-sensitive slicing, published in [61]. In Chapter 4, published in [63], we introduce interpolation for the first time to the area of concolic testing and show its exponential benefits in boosting concolic testing. In Chapter 5, published in [21], we propose a novel technique of “lazy” symbolic execution that outperforms current techniques significantly in the context of interpolation-based program verification. In Chapter 6, presented in [62], we employ symbolic execution to tackle the very practical problem of compressing long loop iterations in error traces and explain the compressed trace to the programmer. Finally, Chapter 7 concludes the thesis. The published work [84] is also a part of our contribution to trace understanding, and is alluded to in the concluding chapter.

## Organisation

In each chapter, we present a symbolic execution and interpolation based algorithm to address a particular problem, and show with evidence how our proposed method is either more powerful than existing techniques or benefits them significantly. Importantly, we also elaborate the challenges that symbolic execution and interpolation themselves face in each setting and how to adapt them for solving each problem. Specifically, the problems addressed in this work are the following:

- **Backward Slicing** (Chapter 3)

In Chapter 3, the first part of which was published in [65], we propose a novel symbolic execution based algorithm to compute static slices. One ground-breaking result of our method is that it produces *exact* slices for loop-free programs. By “exact” we mean that the algorithm guarantees to not produce dependencies from spurious (i.e., non-executable) paths. In other words, our algorithm produces the *smallest possible slice* of a loop-free program for any given slicing criterion, limited only by general theorem proving technology that can deduce the (un)satisfiability of a (first-order logic) formula<sup>4</sup>.

Inspired by the previous result that slicing is more effective when there is path sensitivity, we present in the second part of Chapter 3 (published in [61]) a *transformation* of programs with specified target variables, similar to a slicing criterion. These programs are ready to be analysed by a third-party application that seeks some information about the target variables, such as a verifier seeking a property on them. The transformation embodies a path-sensitive expansion of the program so that infeasible paths can be excluded, and is sliced with respect to the target variables.

---

<sup>4</sup>Of course, this problem is undecidable in general, and so is the exact slicing problem.

Due to path-sensitivity, the slicing is more precise than otherwise. Third-party applications of testing and verification perform substantially better on the transformed program compared to a statically sliced one.

- **Concolic Testing**<sup>5</sup> (Chapter 4)

Recently, to alleviate the problem of manually generating test cases and poor quality of code coverage from random testing, *concolic testing* [97, 49, 18, 16]—a portmanteau of “concrete” and “symbolic”—was proposed. As mentioned in Section 1.1, concolic testing also suffers from path explosion, as there are an exponential number of paths to test.

In Chapter 4, published in [63], we propose a novel algorithm to address path explosion in concolic testing using interpolation. We first show that the typical *modus operandi* of interpolation does not work in concolic testing due to the lack of control of a search order, which in this setting is imposed by the concolic tester. This greatly hinders the formation of interpolants from running test cases. Then, we propose a new method based on subsumption to *accelerate* the formation of interpolants in order to get back the exponential benefits that it is known for. Finally, we show with evidence that our proposed algorithm boosts the coverage of an existing concolic tester significantly.

- **Interpolation-based Verification** (Chapter 5)

In Chapters 3 and 4, we show how powerful symbolic execution with interpolation is in program analysis (slicing) and testing. In both settings, its effectiveness heavily relies on the quality of the computed interpolants, which are the key to mitigating path explosion. Symbolic execution avoids the exploration of *infeasible paths* by stopping the moment infeasibility is encountered in its path condition, a property referred to as being *eager*, and one considered an advantage.

---

<sup>5</sup>Concolic testing is now commonly referred to as “dynamic symbolic execution”. We use the former term for historical reasons.

In Chapter 5, published in [21], we show that in the setting of program verification, being eager is not always beneficial for symbolic execution, as it can hinder the discovery of better interpolants. We present a systematic algorithm that speculates that an infeasibility may be temporarily ignored for the purpose of “learning” better interpolants about the path in question. This speculation is bounded and so does not make symbolic execution lose its intrinsic benefits. We demonstrate using real benchmarks that this “lazy” variant of symbolic execution that ignores infeasibilities outperforms its eager counterpart by a factor of two or more.

- **Trace Understanding** (Chapter 6)

Reasoning about long execution (typically, error) traces is an integral but tedious part of software development, especially in debugging. In Chapter 6, presented in [62], we propose an algorithm to compress execution traces by discovering *loop invariants* for the iterations in the trace. The invariants discovered are “safe”, such that the compressed trace obeys the original trace’s semantics regarding the assertion at the end. Thus, the compressed trace concisely explains the original trace without unrolling the loops fully.

A central feature is the use of a canonical loop invariant discovery algorithm which preserves all atomic formulas in the representation of a symbolic state which can be shown to be invariant. If this fails to provide a “safe” invariant, then the algorithm dynamically unrolls the loop and attempts the discovery at the next iteration, where it is more likely to succeed as the loop stabilises towards an invariant. We show via realistic benchmarks, which present the compressed trace as a Hoare proof, that the end result is significantly more succinct than the original trace.

# Chapter 2

## Preliminaries

Throughout this thesis, we restrict our presentation to a simple imperative programming language where all basic operations are either assignments or assume operations, and the domain of all variables are integers (pointers are treated as indices on a special array representing the heap). The set of all program variables is denoted by *Vars*. An *assignment*  $x = e$  corresponds to the assignment of the evaluation of the expression  $e$  to the variable  $x$ . In the *assume* operator,  $\text{assume}(c)$ , if the Boolean expression  $c$  evaluates to *true*, then the program continues, otherwise it halts. The set of operations is denoted by *Ops*. We then model a program by a *transition system*. A transition system is a quadruple  $\langle \Sigma, I, \longrightarrow, O \rangle$  where  $\Sigma$  is the set of states and  $I \subseteq \Sigma$  is the set of initial states.  $\longrightarrow \subseteq \Sigma \times \Sigma \times Ops$  is the transition relation that relates a state to its (possible) successors. This transition relation models the operations that are executed when control flows from one program location to another. We shall use  $\ell \xrightarrow{\text{op}} \ell'$  to denote a transition relation from  $\ell \in \Sigma$  to  $\ell' \in \Sigma$  executing the operation  $\text{op} \in Ops$ . Finally,  $O \subseteq \Sigma$  is the set of final states.

### 2.1 Symbolic Execution

A *symbolic state*  $v$  is a triple  $\langle \ell, s, \Pi \rangle$ . The symbol  $\ell \in \Sigma$  corresponds to the current program location. For clarity of presentation in our algorithm, we will use special symbols for initial location,  $\ell_{\text{start}} \in I$ , final location,  $\ell_{\text{end}} \in O$ , and bug location  $\ell_{\text{error}} \in O$  (if any). W.l.o.g we assume that there is only one initial,



final, and bug location in the transition system. We shall use a similar notation  $\mathfrak{v} \xrightarrow{\text{op}} \mathfrak{v}'$  to denote a transition from the symbolic state  $\mathfrak{v}$  to  $\mathfrak{v}'$  corresponding to their program locations.

The symbolic store  $s$  is a function from program variables to terms over input symbolic variables. Each program variable is initialised to a fresh input symbolic variable. This is done by the procedure *init\_store()*. The *evaluation*  $\llbracket c \rrbracket_s$  of a constraint expression  $c$  in a store  $s$  is defined recursively as usual:  $\llbracket v \rrbracket_s = s(v)$  (if  $c \equiv v$  is a variable),  $\llbracket n \rrbracket_s = n$  (if  $c \equiv n$  is an integer),  $\llbracket e \text{ op}_r e' \rrbracket_s = \llbracket e \rrbracket_s \text{ op}_r \llbracket e' \rrbracket_s$  (if  $c \equiv e \text{ op}_r e'$  where  $e, e'$  are expressions and  $\text{op}_r$  is a relational operator  $<, >, =, \neq, \geq, \leq$ ), and  $\llbracket e \text{ op}_a e' \rrbracket_s = \llbracket e \rrbracket_s \text{ op}_a \llbracket e' \rrbracket_s$  (if  $c \equiv e \text{ op}_a e'$  where  $e, e'$  are expressions and  $\text{op}_a$  is an arithmetic operator  $+, -, \times, \dots$ ). Sometimes, when the context of usage is clear, we simply say  $\llbracket \mathfrak{v} \rrbracket$  to mean the evaluation of the symbolic state  $\mathfrak{v}$  with its own symbolic store.

Finally,  $\Pi$  is called *path condition*, a first-order formula over the symbolic inputs that accumulates constraints which the inputs must satisfy in order for an execution to follow the particular corresponding path. The set of first-order formulas and symbolic states are denoted by *FOL* and *SymStates*, respectively. Given a transition system  $\langle \Sigma, I, \longrightarrow, O \rangle$  and a state  $\mathfrak{v} \equiv \langle \ell, s, \Pi \rangle \in \text{SymStates}$ , the symbolic execution of  $\ell \xrightarrow{\text{op}} \ell'$  returns another symbolic state  $\mathfrak{v}'$  defined as:

$$\text{SYMSTEP}(\mathfrak{v}, \ell \xrightarrow{\text{op}} \ell') \triangleq \mathfrak{v}' \triangleq \begin{cases} \langle \ell', s, \Pi \wedge \llbracket c \rrbracket_s \rangle & \text{if } \text{op} \equiv \text{assume}(c) \text{ and} \\ & \Pi \wedge \llbracket c \rrbracket_s \text{ is satisfiable} \\ \langle \ell', s[x \mapsto \llbracket e \rrbracket_s], \Pi \rangle & \text{if } \text{op} \equiv x = e \end{cases} \quad (2.1)$$

Note that Equation (2.1) queries a constraint solver for satisfiability checking on the path condition. We assume the solver is sound but not necessarily complete. That is, the solver must say a formula is unsatisfiable only if it is indeed so.

Abusing notation, given a symbolic state  $\mathfrak{v} \equiv \langle \ell, s, \Pi \rangle$  we define  $\llbracket \mathfrak{v} \rrbracket : \text{SymStates} \rightarrow \text{FOL}$  as the formula  $(\bigwedge_{v \in \text{Vars}} \llbracket v \rrbracket_s) \wedge \Pi$  where *Vars* is the set of program variables.

A *symbolic path*  $\pi \equiv \mathfrak{v}_0 \cdot \mathfrak{v}_1 \cdot \dots \cdot \mathfrak{v}_n$  is a sequence of symbolic states such that

$\ell_1$ <b>if</b> ( $x > y$ ) { $\ell_2$ $x = x + y$ ; $\ell_3$ $y = x - y$ ; $\ell_4$ $x = x - y$ ; $\ell_5$ <b>if</b> ( $x - y > 0$ ) $\ell_6$ <code>error()</code> ; } $\ell_7$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;"><math>\ell_1</math>    <code>assume(x &gt; y)</code></td> <td style="width: 50%;"><math>\ell_2</math></td> </tr> <tr> <td><math>\ell_2</math>    <math>x := x + y</math></td> <td><math>\ell_3</math></td> </tr> <tr> <td><math>\ell_3</math>    <math>y := x - y</math></td> <td><math>\ell_4</math></td> </tr> <tr> <td><math>\ell_4</math>    <math>x := x - y</math></td> <td><math>\ell_5</math></td> </tr> <tr> <td><math>\ell_5</math>    <code>assume(x-y &gt; 0)</code></td> <td><math>\ell_6</math></td> </tr> <tr> <td><math>\ell_5</math>    <code>assume(x-y ≤ 0)</code></td> <td><math>\ell_7</math></td> </tr> <tr> <td><math>\ell_1</math>    <code>assume(x ≤ y)</code></td> <td><math>\ell_7</math></td> </tr> </table>	$\ell_1$ <code>assume(x &gt; y)</code>	$\ell_2$	$\ell_2$ $x := x + y$	$\ell_3$	$\ell_3$ $y := x - y$	$\ell_4$	$\ell_4$ $x := x - y$	$\ell_5$	$\ell_5$ <code>assume(x-y &gt; 0)</code>	$\ell_6$	$\ell_5$ <code>assume(x-y ≤ 0)</code>	$\ell_7$	$\ell_1$ <code>assume(x ≤ y)</code>	$\ell_7$
$\ell_1$ <code>assume(x &gt; y)</code>	$\ell_2$														
$\ell_2$ $x := x + y$	$\ell_3$														
$\ell_3$ $y := x - y$	$\ell_4$														
$\ell_4$ $x := x - y$	$\ell_5$														
$\ell_5$ <code>assume(x-y &gt; 0)</code>	$\ell_6$														
$\ell_5$ <code>assume(x-y ≤ 0)</code>	$\ell_7$														
$\ell_1$ <code>assume(x ≤ y)</code>	$\ell_7$														
(a)	(b)														

Figure 2.1: (a) A program to swap two integers (b) Its transition system

$\forall i \bullet 1 \leq i \leq n$  the state  $\nu_i$  is a *successor* of  $\nu_{i-1}$ , denoted as  $\text{SUCC}(\nu_{i-1}, \nu_i)$ <sup>1</sup>. A symbolic state  $\nu' \equiv \langle \ell', \cdot, \cdot \rangle$  is a successor of another  $\nu \equiv \langle \ell, \cdot, \cdot \rangle$  if there exists a transition relation  $\ell \xrightarrow{\text{op}} \ell'$ . A path  $\pi \equiv \nu_0 \cdot \nu_1 \cdot \dots \cdot \nu_n$  is *feasible* if  $\nu_n \equiv \langle \ell, s, \Pi \rangle$  such that  $\llbracket \Pi \rrbracket_s$  is satisfiable. If  $\ell \in O$  and  $\nu_n$  is feasible then  $\nu_n$  is called *terminal* state. Otherwise, if  $\llbracket \Pi \rrbracket_s$  is unsatisfiable the path is called *infeasible* and  $\nu_n$  is called an *infeasible* state. If there exists a feasible path  $\pi \equiv \nu_0 \cdot \nu_1 \cdot \dots \cdot \nu_n$  then we say  $\nu_k$  ( $0 \leq k \leq n$ ) is *reachable* from  $\nu_0$  in  $k$  steps. We say  $\nu''$  is reachable from  $\nu$  if it is reachable from  $\nu$  in some number of steps.

We also define a (partial) function  $\text{MergePoint} : \text{SymStates} \rightarrow \text{SymStates} \times \text{SymStates}$  that, given a symbolic state  $\nu \equiv \langle \ell, \cdot, \cdot \rangle$  if there is an assume statement at  $\nu$  (i.e.,  $\ell$  corresponds to a branch point), returns a tuple  $\langle \nu_1 \equiv \langle \ell', \cdot, \cdot \rangle, \nu_2 \equiv \langle \ell', \cdot, \cdot \rangle \rangle$  such that  $\nu_1$  and  $\nu_2$  are reachable from  $\nu$ , and  $\ell'$  is the *nearest post-dominator* of  $\ell$ . In other words,  $\nu_1$  and  $\nu_2$  are the symbolic states at the merge point reached through the “then” and “else” body respectively.

Finally, a *symbolic execution tree* contains all the execution paths explored during the symbolic execution of a transition system by triggering Equation (2.1). The nodes represent symbolic states and the arcs represent transitions between states.

Let us exemplify symbolic execution with the help of the program in Fig. 2.1(a), taken from [94]. This program poses a verification problem, namely, to verify if the swap of two integers is correct. Verification can be done by exploring the symbolic execution tree and ensuring that the error location  $\ell_{\text{error}}$ , in this case designated to be  $\ell_6$ , is not reachable. In Fig. 2.1(b), we model the

<sup>1</sup>W.l.o.g, we assume each state has at most two successors.

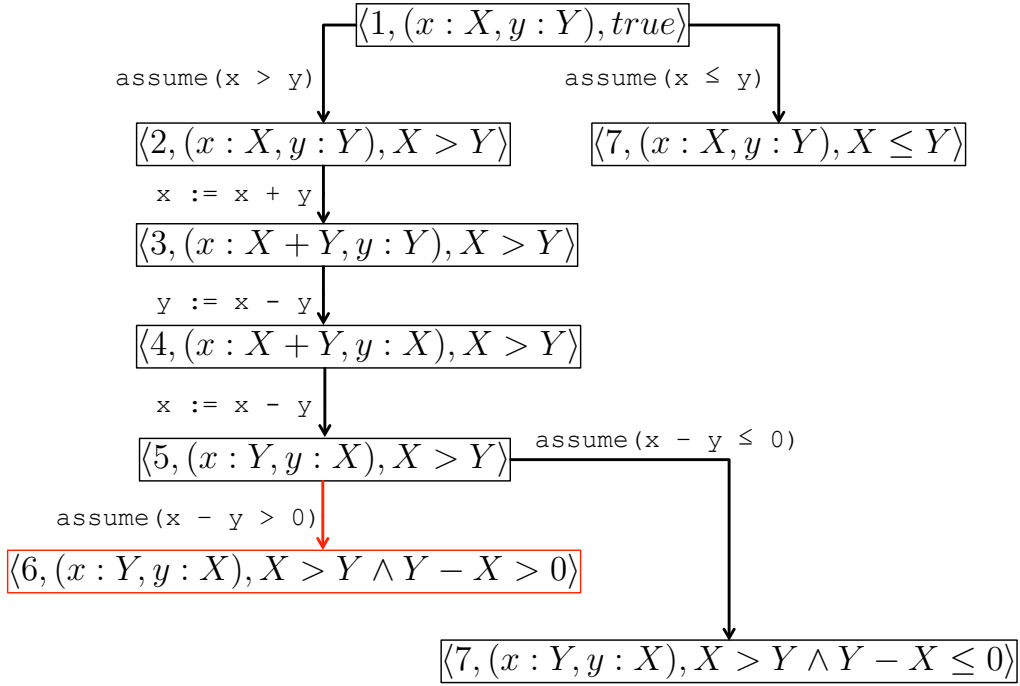


Figure 2.2: Symbolic Execution Tree of the program in Fig. 2.1

program semantics faithfully using our simplified transition system.

The corresponding symbolic execution tree is shown in Fig. 2.2. This tree is obtained by repeated invocations of `SYMSTEP` starting with the initial state  $\langle 1, (x : X, y : Y), true \rangle$ , i.e., the initial program point  $\ell_1$ , variables initialised to arbitrary distinct symbols ( $X$  and  $Y$ ) and the path condition initialised to *true*. Note that at program point  $\ell_5$ , the symbolic store maintains that the variables  $x$  and  $y$  now contain the values  $Y$  and  $X$  respectively, and the path condition declares  $X > Y$ . Therefore, executing the statement `assume(x-y>0)` results in an *infeasible* state at  $\ell_6$  because of the unsatisfiability of the formula  $X > Y \wedge Y - X > 0$ . Thus, the error location  $\ell_6$  is proven to be unreachable.

In general, when a path is symbolically executed, we can extract analysis information from it. In the above case of verification, the “analysis” was to prove the path safe, but we can also compute variable dependency information (slicing), live variable information (liveness analysis), check for bugs along the path (testing), compute its execution time (worst-case execution time analysis), etc. This analysis information can be *annotated* at each symbolic state along the path to allow merging of analysis results from different paths and re-use of previously computed results, which we will see next.

## 2.2 Interpolation and Witnesses

During symbolic execution, if two symbolic states  $\mathfrak{v}$  and  $\mathfrak{v}'$  at a program point  $\ell$  are encountered such that  $\llbracket \mathfrak{v} \rrbracket \equiv \llbracket \mathfrak{v}' \rrbracket$  then exploring  $\mathfrak{v}$  is clearly a waste of effort (assuming  $\mathfrak{v}'$  was already explored). One can instead *merge*  $\mathfrak{v}$  with  $\mathfrak{v}'$  and reuse its already computed solution. Unfortunately, the chances of encountering such  $\mathfrak{v}$ s are highly unlikely in practice.

Interpolation and witnesses help to increase this likelihood by discarding some irrelevant information when comparing  $\mathfrak{v}$  and  $\mathfrak{v}'$ . We use the notion of *state interpolation* introduced in [66] and representative witness paths introduced in [67]. The idea is to merge certain symbolic states with another and *reuse* the pre-computed analysis result in a sound and precise manner, provided certain merging conditions are met. We now formalise these conditions.

**Definition 1** (Interpolant). *Given a pair of first order logic formulas  $A$  and  $B$  such that  $A \wedge B$  is false, an interpolant [28]  $INTP(A,B)$  is another formula  $\bar{\Psi}$  such that*

- (a)  $A \models \bar{\Psi}$ ,
- (b)  $\bar{\Psi} \wedge B$  is false, and
- (c)  $\bar{\Psi}$  is formed using common variables of  $A$  and  $B$ .

Interpolation allows us to remove irrelevant facts from  $A$  without affecting the unsatisfiability of  $A \wedge B$ . Whenever an infeasible path is met, with  $A$  being the path “prefix” and  $B$  being typically the last encountered guard, the interpolant succinctly captures the reason of infeasibility of the path, discarding irrelevant information from the path condition. An interpolant is then generated at the state of infeasibility and is propagated back through the path to be generated at each state. Efficient interpolation algorithms exist for quantifier-free fragments of theories such as linear real/integer arithmetic, uninterpreted functions, pointers and arrays, and bitvectors (e.g., see [23] for details) where interpolants can be extracted from the refutation proof in linear time on the size of the proof.

For instance, in Fig. 2.2, at program point 5,  $A$  is the formula  $Y_0 > X_0$  and  $B$  is the formula  $X_0 - Y_0 > 0$ , where  $X_0$  and  $Y_0$  are the initial symbolic values of  $x$  and  $y$ . These are obtained from evaluating the path conditions at program

point 5 and the branch condition at program point 6, respectively. One possible interpolant is the formula  $X_0 - Y_0 \leq 0$ , which when propagated to program point 4 would become  $X_0 - 2 \times Y_0 \leq 0$  (it may appear that we are doing a *weakest pre-condition* computation; indeed, as we will see in later, it is one of the ways to compute interpolants).

**Definition 2** (Witness Paths and Formulas). *Given a symbolic state  $\nu \equiv \langle \ell, \cdot, \cdot \rangle$  that is annotated with the analysis result  $\sigma_\nu$ , a witness path, represented as  $\omega_\nu$ , is a representative feasible symbolic path  $\pi \equiv \nu \cdot \dots \cdot \nu_{\text{end}}$  from which the analysis result  $\sigma_\nu$  is derived.*

Intuitively, a witness path at a symbolic state  $\nu$  is a representative path through the state from which the analysis result at  $\nu$  was obtained. Note that there may be a large number of paths passing through  $\nu$  but the analysis result at  $\nu$  could have been obtained only from a few.

For instance, in the case of slicing, the analysis information stored at each state would be the set of dependency variables (i.e., variables at this state that affect the target “slicing criterion”). In this case, a witness path for a dependency variable  $x$  at  $\nu$  is a path along which  $x$  affects the slicing criterion at the end. In general there may be more than one witness path for a particular symbolic state.

Now, using the notions of interpolants and witness paths, we define our most important condition for merging two symbolic states:

**Definition 3** (Merging Conditions). *Given a current symbolic state  $\nu \equiv \langle \ell, s, \Pi \rangle$  and an already annotated symbolic state  $\nu' \equiv \langle \ell, s', \Pi' \rangle$  such that  $\bar{\Psi}_{\nu'}$  is an interpolant generated for  $\nu'$ ,  $\sigma_{\nu'}$  is the analysis result for  $\nu'$ , and  $\omega_{\nu'}$  is the witness path at  $\nu'$ , we say  $\nu$  can be merged with  $\nu'$  if the following conditions hold:*

$$\begin{aligned} (a) \quad \llbracket \nu \rrbracket &\models \bar{\Psi}_{\nu'} \\ (b) \quad \omega_{\nu'} &\text{ is feasible from } \nu \end{aligned} \tag{2.2}$$

Note importantly that both  $\nu$  and  $\nu'$  must correspond to the same program point  $\ell$  in order to be merged. Once  $\nu$  is merged with  $\nu'$ , symbolic execution of  $\nu$  can

simply stop, and the analysis result  $\sigma_{\mathfrak{v}'}$  from  $\mathfrak{v}'$  can be reused at  $\mathfrak{v}$ .

The condition (a) of Eqn. 2.2, also called the “subsumption check”, affects *soundness* and it ensures that the set of feasible symbolic paths reachable from  $\mathfrak{v}$  is a subset of those from  $\mathfrak{v}'$ . This is a necessary condition for two states to be merged, formalised below.

**Lemma 1.** *Given states  $\mathfrak{v} \equiv \langle \ell, s, \Pi \rangle$  and  $\mathfrak{v}' \equiv \langle \ell, s', \Pi' \rangle$ , let  $\bar{\Psi}_{\mathfrak{v}'}$  be the interpolant for  $\mathfrak{v}'$ . If  $\llbracket \mathfrak{v} \rrbracket \models \bar{\Psi}_{\mathfrak{v}'}$ , the set of feasible paths from  $\mathfrak{v}$  is a subset of those from  $\mathfrak{v}'$ .*

PROOF. (By contradiction). Assume there exists a feasible path  $\pi$ , with path condition  $\Pi_\pi$ , from  $\mathfrak{v}$  but  $\pi$  is infeasible from  $\mathfrak{v}'$ . If  $\pi$  is infeasible from  $\mathfrak{v}'$  then  $\llbracket \mathfrak{v}' \rrbracket \wedge \Pi_\pi$  is unsatisfiable, and by definition of interpolant,  $\bar{\Psi}_{\mathfrak{v}'} \wedge \Pi_\pi$  is unsatisfiable. Since  $\llbracket \mathfrak{v} \rrbracket \models \bar{\Psi}_{\mathfrak{v}'}$ , it follows that  $\llbracket \mathfrak{v} \rrbracket \wedge \Pi_\pi$  is unsatisfiable. However, since  $\pi$  is feasible from  $\mathfrak{v}$ ,  $\llbracket \mathfrak{v} \rrbracket \wedge \Pi_\pi$  cannot be unsatisfiable.  $\square$

To understand the intuition behind the subsumption check, it helps to know what an interpolant at a node actually represents. An interpolant  $\bar{\Psi}_{\mathfrak{v}'}$  at a node  $\mathfrak{v}'$  succinctly captures the reason of infeasibility of all infeasible paths in the symbolic tree rooted at  $\mathfrak{v}'$ . Let us call this tree  $T_1$ . Then, if another state  $\mathfrak{v}$  at  $\ell$  is encountered such that  $\llbracket \mathfrak{v} \rrbracket \models \bar{\Psi}_{\mathfrak{v}'}$ , it means that any infeasible path in  $T_1$  is also infeasible in the tree rooted at  $\mathfrak{v}$ , say  $T_2$ . In other words, any feasible path in  $T_2$  is also feasible under  $T_1$ . Thus, the analysis information derived from  $T_1$  is a *sound* approximation of the analysis information about  $T_2$ .

The condition (b) of Eqn. 2.2 is the witness check which affects *accuracy* and ensures that the merging of two states does not incur any loss of precision. This is formalised in the following theorem.

**Theorem 1.** *Given states  $\mathfrak{v} \equiv \langle \ell, s, \Pi \rangle$  and  $\mathfrak{v}' \equiv \langle \ell, s', \Pi' \rangle$ , let  $\sigma_{\mathfrak{v}'}$  be the analysis result associated with  $\mathfrak{v}'$ . If  $\mathfrak{v}$  can be merged with  $\mathfrak{v}'$  by satisfying the conditions of Eqn. 2.2, then by exploring  $\mathfrak{v}$  there cannot be produced an analysis result  $\sigma_{\mathfrak{v}}$  such that  $\sigma_{\mathfrak{v}} \neq \sigma_{\mathfrak{v}'}$ .*

Theorem 1 guarantees that had one explored  $\mathfrak{v}$  instead of merging (and reusing) with  $\mathfrak{v}'$ , one would obtain exactly the same analysis information as  $\sigma_{\mathfrak{v}'}$ . The proof for Theorem 1 is given in Chapter 3, in the context of program slicing.

A subtle point is that witnesses are not required if the “analysis” being performed is program verification and testing. Typically, in verification and testing, there are only two results that an analysis of a tree can provide: “safe” or “unsafe”. If the result at any state is “unsafe”, the process generally terminates. Otherwise, soundness of reuse—dictated by condition (a) of Eqn. 2.2 using interpolants—automatically guarantees precision of reuse, as “safe” is trivially the most precise result. This important observation allows for optimisations in the implementation of our algorithms for verification and testing.

## Interpolant strength and Weakest Preconditions

It is easy to see that the *weaker* the interpolant is in logical strength, the more likely it is to subsume other states, as it would have filtered away more irrelevant information. Ideally, the *weakest precondition* [55] (WP), denoted by  $\widehat{wlp}$ , at a symbolic state  $\upsilon$  (w.r.t. the infeasibility of paths that pass through  $\upsilon$ ) is the perfect interpolant. Since WP is computationally expensive, we under-approximate it by a mix of existential quantifier elimination, unsatisfiable cores, and some heuristics. Whenever an infeasible path is detected we compute  $\neg(\exists \bar{y} \cdot G)$ , the *postcondition* that we want to map into a *precondition*, where  $G$  is the guard where the infeasibility is detected and  $\bar{y}$  are  $G$ -local variables. The two main rules for propagating WP’s are:

$$(A) \widehat{wlp}(x := e, Q) = Q[e/x]$$

$$(B) \widehat{wlp}(\mathbf{if}(C) S1 \mathbf{else} S2, Q) = (C \Rightarrow \widehat{wlp}(S1, Q)) \wedge (\neg C \Rightarrow \widehat{wlp}(S2, Q))$$

Rule (A) replaces all occurrences of  $x$  with  $e$  in the formula  $Q$ . Rule (B), the standard WP propagation rule for branch points, poses a problem as it makes the formula *disjunctive* and grow exponentially in size. The challenge is to produce non-disjunctive formulas from rule (B) but still as weak as possible to increase the likelihood of subsumption. To tackle this, during forward symbolic execution when an infeasible path is detected we discard *irrelevant* guards by using *unsatisfiable cores* (UC) to avoid growing the WP formula unnecessarily.

**Definition 4** (Unsatisfiable Core). *Given a constraint set  $S$  whose conjunction is unsatisfiable, an unsatisfiable core (UC)  $S'$  is any unsatisfiable subset of  $S$ . An unsatisfiable core  $S'$  is minimal if any strict subset of  $S'$  is satisfiable.*

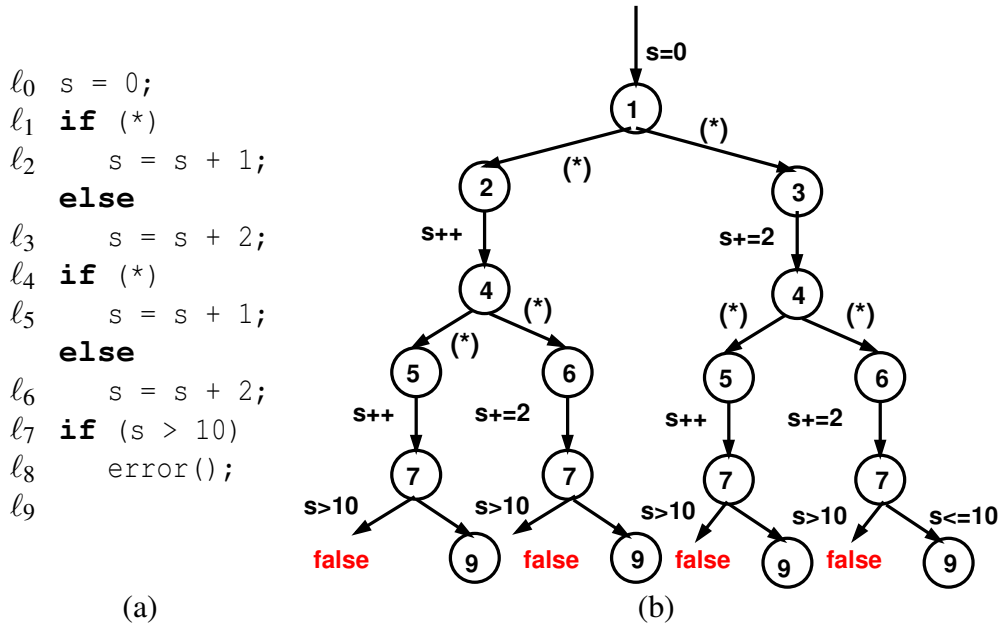


Figure 2.3: (a) A verification problem (b) Its full symbolic execution tree

For instance, the formula  $C \Rightarrow \widehat{wlp}(S1, Q)$  can be replaced with  $\widehat{wlp}(S1, Q)$  if  $C \notin \mathcal{C}$  where  $\mathcal{C}$  is a (not necessarily minimal) UC. Otherwise, we underapproximate  $C \Rightarrow \widehat{wlp}(S1, Q)$  as follows. Let  $d_1 \vee \dots \vee d_n$  be  $\neg \widehat{wlp}(S1, Q)$  then we compute  $\bigwedge_{1 \leq i \leq n} (\neg (\exists \bar{x}^i. (C \wedge d_i)))$ , where existential quantifier elimination removes the post-state variables  $\bar{x}^i$ . A very effective heuristic if the resulting formula is disjunctive is to delete those conjuncts that are not implied by  $C$  because they are more likely to be irrelevant to the infeasibility reason.

Let us now exemplify the use of interpolation during symbolic execution. Consider the program in Fig. 2.3(a) (taken from [64]), where a  $*$  represents an operation that returns a non-deterministic outcome (*true* or *false*). The program initialises a variable  $s$  to 0, and performs two increments of either 1 or 2. The safety property checks whether the value of  $s$  is greater than 10, and if so, an error is thrown. The full symbolic execution tree<sup>2</sup> of this program is shown in Fig. 2.3(b). Clearly the program is safe, as the error location  $l_8$  is never reached. However, the symbolic execution tree is exponential in the number of branches.

Suppose that we symbolically executed the program with interpolation. After executing the first path, as shown in Fig. 2.4(a), we would annotate each program point along the path with WP interpolants (starting from  $\bar{\Psi}_7$  and prop-

<sup>2</sup>From now on, for clarity, we do not show the symbolic states explicitly as in Fig. 2.2, but rather only the program points and transitions.



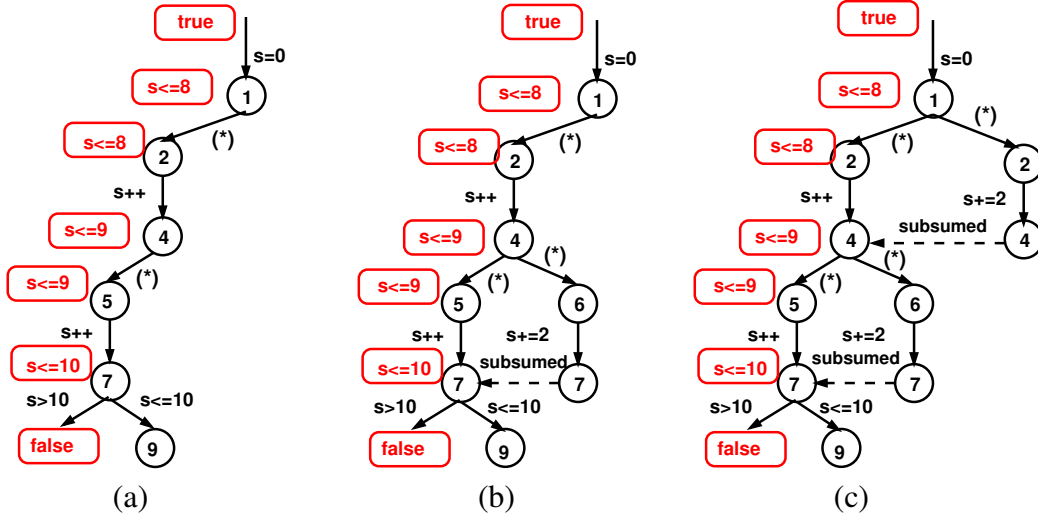


Figure 2.4: Building the Symbolic Execution Tree with Interpolation (WP)

agating backwards through the assignments):  $\bar{\Psi}_7 : s \leq 10$ ,  $\bar{\Psi}_5 : s \leq 9$ ,  $\bar{\Psi}_4 : s \leq 9$ ,  $\bar{\Psi}_2 : s \leq 8$ , and  $\bar{\Psi}_1 : s \leq 8$ . Note that the interpolant at a point is to be interpreted on the latest version of the program variables at that point.

In this example, the WP computations are notably simplified since the guards are clearly irrelevant for the infeasibility of the path, and hence, only rule (A) is triggered. For instance,  $\bar{\Psi}_7 : s \leq 10$  is obtained by  $\neg(\exists \mathcal{V} \setminus \{s\} \cdot s > 10) \equiv s \leq 10$  where  $\mathcal{V}$  is the set of all program variables (including renamed variables), and  $\bar{\Psi}_6 : s \leq 9$  is obtained by  $\widehat{wlp}(s' = s + 1, s' \leq 10) = s \leq 9$ , where  $s$  and  $s'$  are the pre-state and post-state variables. Fig. 2.4(b) shows the second symbolic path but note that the path can be now subsumed at location 7 since the symbolic state  $s = 0 \wedge s' = s + 1 \wedge s'' = s' + 2 \models s'' \leq 10$ . Dashed edges represent subsumed paths and are labelled with “subsumed”. Finally, Fig. 2.4(c) illustrates how the third symbolic path can be also subsumed at location 4 since  $s = 0 \wedge s' = s + 2 \models s' \leq 9$ . Now, we have managed to prove safety again but the size of the symbolic tree is linear on the number of branches.

This example shows that interpolation can result in exponential savings. However, it is important to note that this benefit is greatly affected by the *quality* of interpolants. For instance, had we used strongest postcondition (SP) interpolants for proving the above program safe, we might not have obtained a linear tree. In this thesis, we use either WP or SP interpolants in our examples, although there are also numerous other interpolation methods. Hence we make

it clear that *the actual interpolation method used is orthogonal to this work.*

## 2.3 Implementation: TRACER

This section presents TRACER, the framework that we developed for performing symbolic execution with interpolation, and provides implementation details and tricks for anyone who may be interested in implementing the algorithms in this thesis. TRACER was originally presented in [64] as a verifier of safety properties of C programs, and can be downloaded from [2].

Essentially, TRACER implements classical symbolic execution [71] with some novel features that we will outline along this section. It takes symbolic inputs rather than actual data and executes the program considering those symbolic inputs. During the execution of a path all its constraints are accumulated in a first-order logic (FOL) formula called *path condition (PC)*. Whenever code of the form `if(C) then S1 else S2` is reached the execution forks the current symbolic state and updates path conditions along both the paths:  $PC_1 \equiv PC \wedge C$  and  $PC_2 \equiv PC \wedge \neg C$ . Then, it checks if either  $PC_1$  or  $PC_2$  is unsatisfiable. If yes, then the path is *infeasible* and the execution halts backtracking to the last choice point. Otherwise, it follows the path.

The first key aspect of TRACER, originally proposed in [66] for symbolic execution, is the avoidance of full enumeration of symbolic paths by *learning* from infeasible paths computing *interpolants* [28]. Preliminary versions of TRACER [59, 66] computed interpolants based on *strongest postconditions*. Given two formulas  $A$  (symbolic path) and  $B$  (last guard where infeasibility is detected) such that  $A \wedge B$  is unsat, an interpolant was obtained by  $\exists \bar{x} \cdot A$  where  $\bar{x}$  are  $A$ -local variables (i.e., variables occurring only in  $A$ ). However, as mentioned in Section 2.2, weaker interpolants favour better subsumption, and hence we implemented the weakest precondition approximation method explained before. Having said that, we have indeed encountered benchmarks in practice where interpolants based on strongest postconditions were good enough and faster to compute than weakest preconditions.

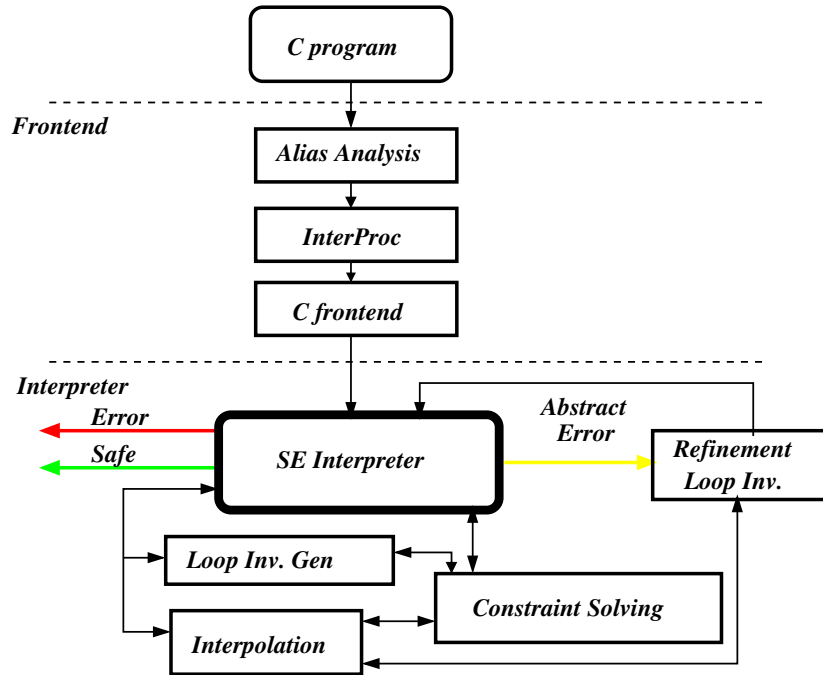


Figure 2.5: Architecture of TRACER

## Usage and Architecture of TRACER

**Input.** TRACER takes as input a C program with either assertions of the form `_TRACER_abort(Cond)`, where *Cond* is a quantifier-free FOL formula, or annotations of the form `_TRACER_slice(Vars)`, where *Vars* is a set of variables to slice on. In the former case (i.e., verification or testing), each path that encounters the assertion tests whether *Cond* holds or not. If yes, the symbolic execution has reached an error node and thus, it reports the error and aborts if the error is real, or refines if spurious. Otherwise, the symbolic execution continues normally. In the latter case (i.e., slicing mode), each path that encounters the `_TRACER_slice` annotation is used to compute backward dependency information.

**Output.** If in verification or testing mode, the symbolic execution terminates and all `_TRACER_abort` assertions failed then the program is reported as safe and the corresponding symbolic execution tree is displayed as the proof object. If the program is unsafe then a counterexample is shown. In slicing mode, once the symbolic execution terminates computing dependency information at all nodes, the slice is computed using this information.

**Architecture.** Fig. 2.5 outlines the architecture of TRACER. It is divided into two components. First, a C-frontend based on CIL [85] translates the program

into a constraint-based logic program. Both pointers and arrays are modeled using the theory of arrays. An alias analysis is used in order to yield sound and finer grained independent partitions (i.e., *separation*) as well as infer which scalars' addresses may have been taken. Optionally, INTERPROC [74] (option `-loop-inv`) can be used to provide loop invariants. The second component is an interpreter which symbolically executes the constraint-based logic program and it aims at demonstrating that error locations are unreachable. This interpreter is implemented in a *Constraint Logic Programming (CLP)* system called  $\text{CLP}(\mathcal{R})$  [60]. Its main sub-components are:

- *Constraint Solving* relies on the  $\text{CLP}(\mathcal{R})$  solver to reason fast over linear arithmetic over reals augmented with a decision procedure for arrays (option `-mccarthy`).
- *Interpolation* is implemented in TRACER by two methods with different logical strength. The first method uses *strongest postconditions* [59, 66] (`-intp sp`). The second computes *weakest preconditions* (`-intp wp`) but currently it only supports linear arithmetic over reals. TRACER also provides interfaces to other interpolation methods such as CLP-PROVER [92] (`-intp clp`).
- *Unbounded Loops* are handled by TRACER using the technique described in [59]. With unbounded loops the only hope to produce a proof is *abstraction*. In a nutshell, upon encountering a cycle TRACER computes the *strongest* possible loop invariants  $\bar{\Psi}$  by using widening techniques in order to make the SE finite. If a spurious abstract error is found then a *refinement phase* (similar to CEGAR [24] methods) discovers an interpolant  $I$  that rules the spurious error out. After restart, TRACER strengthens  $\bar{\Psi}$  by conjoining it with  $I$  and the symbolic execution checks *path by path* if the new strengthened formula is loop invariant. If this test fails for a path  $\pi$ , then TRACER *unrolls*  $\pi$  one more iteration and continues with the process. Notice that the generation of invariants is *dynamic* in the sense that loop unrolls will expose new constraints producing new invariant candidates.

# Chapter 3

## Backward Slicing

Backward slicers are typically path-insensitive (i.e., they ignore the evaluation of predicates in guards), or they are only partially path-sensitive sometimes producing too big slices. Though the value of path-sensitivity is always desirable, as mentioned in Chapter 1, the major challenge is that there are, in general, an exponential number of predicate combinations to be considered.

We make two contributions to the area of backward slicing. Firstly, in Part I of this chapter, we present a *path-sensitive* backward slicer and demonstrate its practicality with real C programs. The core is a symbolic execution-based algorithm that excludes spurious dependencies lying on infeasible paths while pruning paths that cannot improve the accuracy of the dependencies already computed by other paths.

Secondly, in Part II of this chapter, we present a *program transformation* technique intended for programs that are about to be processed by third-party applications querying target variables, such as a verifier or tester. The transformation embodies two concepts – path-sensitivity to exclude infeasible paths, and slicing with respect to the target variables. This key step is founded on a novel idea introduced in this work, called “Tree Slicing”. Compared to the original program, the transformed program may be bigger (due to path-sensitivity) or smaller (due to slicing). We show that it is not much bigger in practice, if at all. The main result however concerns its quality: third-party testers and verifiers perform substantially better on the transformed program compared to the original.

# Part I: Static Backward Slicing

Weiser [104] defined the *backward slice* of a program with respect to a program location  $\ell$  and a variable  $x$ , called the slicing criterion, as all statements of the program that might affect the value of  $x$  at  $\ell$ , considering all possible executions of the program. Slicing was first developed to facilitate software debugging, but it has subsequently been used for performing diverse tasks such as parallelisation, software testing and maintenance, program comprehension, reverse engineering, program integration and differencing, and compiler tuning.

Although static slicing has been successfully used in many software engineering applications, slices may be quite imprecise in practice - "*slices are bigger than expected and sometimes too big to be useful [10]*". Two possible sources of imprecision are: inclusion of dependencies originated from *infeasible paths*, and merging abstract states (via join operator) along incoming edges of a *control flow merge*. A systematic way to avoid these inaccuracies is to perform path-sensitive analysis. An analysis is said to be *path-sensitive* if it keeps track of different state values based on the evaluation of the predicates at conditional branches. Path-sensitive analyses are very rare due to the difficulty of designing efficient algorithms that can handle their combinatorial nature.

The main result of this work is a practical path-sensitive algorithm to compute backward slices. *Symbolic execution (SE)* is the underlying technique that provides path-sensitiveness to our method. The idea behind SE is to use symbolic inputs rather than actual data and execute the program considering those symbolic inputs. During the execution of a path all its constraints are accumulated in a formula  $P$ . Whenever code of the form `if(C) then S1 else S2` is reached the execution forks the current state and updates the two copies  $P_1 \equiv P \wedge C$  and  $P_2 \equiv P \wedge \neg C$ , respectively. Then, it checks if either  $P_1$  or  $P_2$  is unsatisfiable. If yes, then the path is *infeasible* and hence, the execution stops and backtracks to the last choice point. Otherwise, the execution continues. The set of all paths explored by symbolic execution is called the *symbolic execution tree (SET)*.

Not surprisingly, a backward slicer can be easily adapted to compute slices on SETs rather than control flow graphs (CFGs) and then mapping the results

from the SET to the original CFG. It is not difficult to see that the result would be a fully path-sensitive slicer. However, there are two challenges facing this idea. First, the *path explosion problem* in path-sensitive analyses that is also present in SE since the size of the SET is exponential in the number of conditional branches. The second challenge is the infinite length of symbolic paths due to loops. To overcome the latter we borrow from [98] the use of inductive invariants produced from an abstract interpreter to automatically compute *approximate loop invariants*. Because invariants are approximate our algorithm cannot be considered fully path-sensitive in the presence of loops. Nevertheless our results in Sec. 3.4 demonstrate that our approach can still produce significantly more precise slices than a path-insensitive slicer.

Therefore, the main technical contribution of this chapter is how to tackle the path-explosion problem. We rely on the observation that *many symbolic paths have the same impact on the slicing criterion*. In other words, there is no need to explore all possible paths to produce the most precise slice. Our method takes advantage of this observation and explores the search space by dividing the problem into smaller sub-problems which are then solved recursively. Then, it is common for many sub-problems to be “*equivalent*” to others. When this is the case, those sub-problems can be skipped and the search space can be significantly reduced with exponential speedups. In order to successfully implement this search strategy we need to (a) store the solution of a sub-problem as well as the conditions that must hold for reusing that solution, (b) reuse a stored solution if a new encountered sub-problem is “*equivalent*” to one already solved

Our approach symbolically executes the program in a depth-first search manner. This allows us to define a sub-problem as any subtree contained in the SET. Given a subtree, our method following Weiser’s algorithm computes dependencies among variables that allow us to infer which statements may affect the slicing criterion. The fundamental idea for reusing a solution is that when the set of feasible paths in a given subtree is *identical* to that of an already explored subtree, it is not possible to deduce more accurate dependencies from the given subtree. In such cases we can safely reuse dependencies from the explored subtree. However, this check is impractical because it is tantamount to actually

exploring the given subtree, which defeats the purpose of reuse. Hence we define certain reusing conditions, the cornerstone of our algorithm, which are both sound and precise enough to allow reuse without exploring the given subtree.

First, we store a formula that succinctly captures all the infeasible paths detected during the symbolic execution of a subtree. We use efficient *interpolation* techniques [28] to generate *interpolants* for this purpose. Then, whenever a new subtree is encountered we check if the constraints accumulated *imply* in the logical sense the *interpolant* of an already solved subtree. If not, it means there are paths in the new subtree which were unexplored (infeasible) before, and so we need to explore the subtree in order to be sound. Otherwise, the set of paths in the new subtree is a *subset* of that of the explored subtree. However, being a subset is not sufficient for reuse since we need to know if they are *equivalent*, but the equivalence test, as mentioned before, is impractical. Here, we make use of our intuition that only few paths contribute to the dependency information in every subtree. Hence, to check for equivalence of subtrees we need not check all paths, but only those that contribute to the dependencies, what we call the *witness paths* (cf. Fig. 3.1 example). Now, if the implication succeeds we also check if the witness paths of the explored subtree are feasible in the new subtree. If yes, we reuse dependencies. Otherwise, the equivalence test failed.

Finally, as we will discuss in Sec. 3.5, some previous works have tackled the problem of path-sensitive backward slicing before. However, to the best of our knowledge either they suffer from the path-explosion problem or scalability is achieved at the expense of losing some path-sensitiveness. One essential result of our method is that it produces *exact* slices for loop-free programs. By “exact” we mean that the algorithm guarantees to not produce dependencies from spurious<sup>1</sup> (i.e., non-executable) paths. In other words, it produces the *smallest* possible, *sound* slice of a loop-free program for any given slicing criterion. Our method mitigates the path-explosion problem using a combination of interpolants and witness paths that allows pruning significantly the search space.

---

<sup>1</sup>Of course, limited by theorem prover technology which decides whether a formula is unsatisfiable or not.



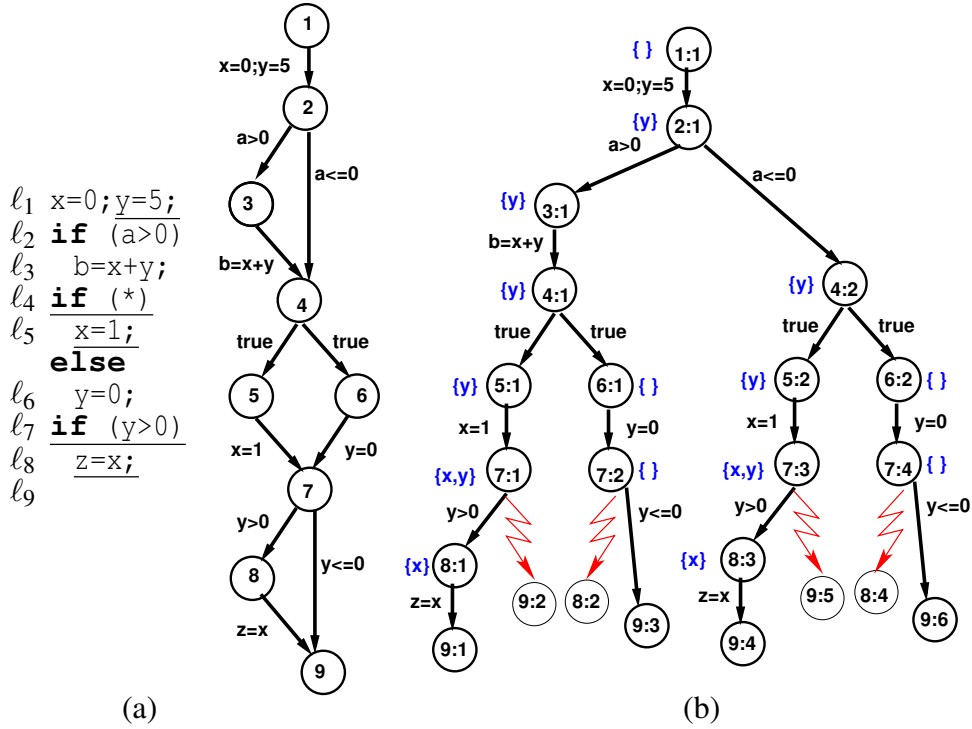


Figure 3.1: (a) A program and its transition system, (b) its naive symbolic execution tree (SET) for slicing criterion (underlined statements)  $\langle \ell_9, \{z\} \rangle$

### 3.1 Motivating Example

We first describe our approach through an example. Consider the program in Fig. 3.1(a) and assume we would like to slice it w.r.t. location  $\ell_9$  and variable  $z$ . The statement  $x=0$  at  $\ell_1$  should not be included in the slice because any path that reaches  $\ell_8$  through  $\ell_5$  redefines  $x$  and any path that reaches  $\ell_8$  through  $\ell_6$  (without redefining  $x$ ) is infeasible. Note that a path insensitive algorithm would **not** be able to infer this from the CFG.

Fig. 3.1(b) shows the naive symbolic execution tree of the program. The nodes are labelled with  $\ell : k$  ( $\ell$  is a program location and  $k$  is an identifier to distinguish nodes with the same program location belonging to different symbolic paths) and edges between two locations are labelled by the intervening program operation. Black (solid) edges denote feasible transitions and red (zigzag) edges denote infeasible transitions. Each node is annotated with its *dependency set* in blue (between brackets) obtained by running, for instance, Weiser’s [104] algorithm. Informally, a dependency set at location  $\ell$  contains all variables that may affect the slicing criterion from any path reachable from  $\ell$ . A statement

at  $\ell$  is included in the slice if the intersection between the dependency set at  $\ell$  and the set of variables defined at  $\ell$  (i.e., left-hand side of the assignment) is not empty. Note that the dependency set at 2:1 only contains  $y$  and therefore, the statement  $x=0$  at  $\ell_1$  would not be included in the slice. Hence it is clear that the path-sensitive SET improves the accuracy of slices. The problem is that the size of the tree is exponential in the number of branches.

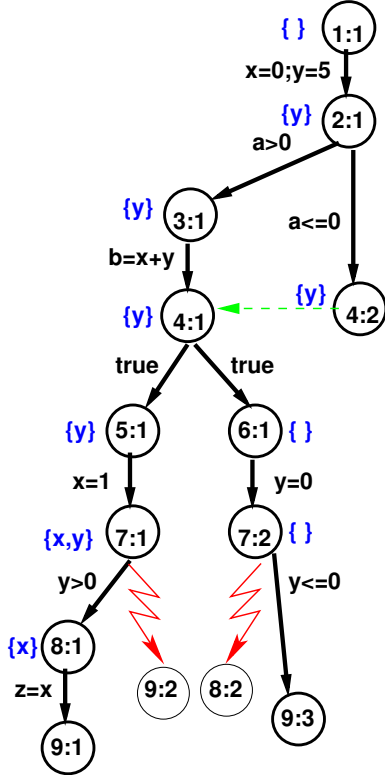


Figure 3.2: Interpolation-based Symbolic Execution Tree for Fig. 3.1

For the above path,  $\Pi_{9:1} \equiv x = 0 \wedge y = 5 \wedge a > 0 \wedge b = x + y \wedge x' = 1 \wedge y > 0 \wedge z = x'$  is the formula built at 9:1, which is satisfiable. It then applies Weiser’s algorithm to compute the dependency set at each node along the path. Briefly, Weiser’s algorithm computes dependency sets starting backwards from the slicing criterion. Whenever an assignment statement modifies a variable appearing in the dependency set following it, the variables in the RHS are added to the dependency set due to *data-dependency*. If the statement is within a branch, the variables in the branch condition are added to the dependency set at the branch

<sup>2</sup>In fact, it is a Directed Acyclic Graph (DAG) due to the existence of reusing edges.

However, consider now the tree in Fig. 3.2 constructed by our method<sup>2</sup> where green (dotted) edges denote reusing transitions. This tree contains the same relevant information needed to exclude  $x=0$  from the slice but without some redundant paths present in Fig. 3.1(b), achieving exponential savings. Let us see how the tree in Fig. 3.2 is generated.

Our algorithm performs symbolic execution guided by depth-first search exploring first the path  $\pi \equiv \ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_7 \cdot \ell_8 \cdot \ell_9$ . As usual, it accumulates the constraints along the path in a formula  $\Pi$ , where variable redefinitions are denoted by primed versions.

point due to *control-dependency*.

In addition to this, our algorithm also computes at each node one of the reusing conditions: the (smallest possible) set of paths from which the dependency set was generated. For example, at 7:1 the dependency set  $\{x, y\}$  was obtained from the path  $\ell_7 \cdot \ell_8 \cdot \ell_9$ , at 4:1 the dependency set  $\{y\}$  was obtained from  $\ell_4 \cdot \ell_5 \cdot \ell_7 \cdot \ell_8 \cdot \ell_9$ , and so on. These paths are called the *witness paths* and they represent the paths along which each variable in the dependency set affects the slicing criterion.

Next our algorithm backtracks and explores the path  $\pi \equiv \ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_7 \cdot \ell_9$  with constraints  $\Pi_{9:2} \equiv x = 0 \wedge y = 5 \wedge a > 0 \wedge b = x + y \wedge x' = 1 \wedge y \leq 0$ . This formula is unsatisfiable and hence the path is infeasible. Now it generates another reusing condition: an *interpolant* that captures the essence of the reason of infeasibility of the path. The main purpose of the interpolant is to exclude irrelevant facts pertaining to the infeasibility so that the reusing conditions are more likely to be reusable in future. For the above path a possible interpolant is  $y = 5$  which is enough to capture its infeasibility and the infeasibility of any path that carries the constraint  $y \leq 0$ <sup>3</sup>. In summary, our algorithm generates two reusing conditions: witness paths from feasible paths and interpolants from infeasible paths.

Next it backtracks and explores the path  $\pi \equiv \ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_6 \cdot \ell_7$ . At 7:2, it checks whether it can reuse the solution from 7:1 by checking if the accumulated constraints  $\Pi_{7:2} \equiv x = 0 \wedge y = 5 \wedge a > 0 \wedge b = x + y \wedge y' = 0$  *imply* the interpolant at 7:1,  $y' = 5$ <sup>4</sup>. Since the implication fails, it has to explore 7:2 in order to be sound. The subtree after exploring this can be seen in Fig. 3.2. Importantly, note here that while applying Weiser's algorithm, it has obtained a more accurate dependency set (empty set) at 7:2 than that which would have been obtained if it reused the solution from 7:1. Also note that at 4:1, the dependency set is still  $\{y\}$  with witness path  $\ell_4 \cdot \ell_5 \cdot \ell_7 \cdot \ell_8 \cdot \ell_9$  and interpolant  $y = 5$ .

Now, when our algorithm backtracks to explore the path  $\pi \equiv \ell_1 \cdot \ell_2 \cdot \ell_4$ , it checks at 4:2 if it can reuse the solution from 4:1. This time, the accumulated constraints  $x = 0 \wedge y = 5 \wedge a \leq 0$  *imply* the interpolant at 4:1,  $y = 5$ . In addition,

<sup>3</sup>For simplicity, we are using strongest postcondition based interpolants here instead of WP.

<sup>4</sup>The interpolant always considers the latest versions of the variables.

the witness path at 4:1 is also feasible under 4:2. Hence, it simply reuses the dependency set  $\{y\}$  from 4:1 both in a *sound* and *precise* manner, and backtracks without exploring 4:2. In this way, it achieves exponential savings while still maintaining as much as accuracy as the naive SET in Fig. 3.1(b). Now, when Weiser’s algorithm propagates back the dependency set  $\{y\}$  from 4:2, we get the dependency set  $\{y\}$  again at 2:1, and the statement  $x=0$  at 1:1 is not included in the slice.

## 3.2 Background

**Program Slicing via Abstract Interpretation.** The *backward slice* of a program w.r.t. a program location  $\ell$  and a set of variables  $V \subseteq \text{Vars}$ , called the *slicing criterion*  $\langle \ell, V \rangle$ , is all statements of the program that might affect the values of  $V$  at  $\ell$ .<sup>5</sup> We follow the dataflow approach described by Weiser [104] reformulated as an abstract domain  $\mathcal{D} \equiv \{\perp\} \cup \mathcal{P}(\text{Vars})$  (where  $\mathcal{P}(\text{Vars})$  is the powerset of program variables) with a lattice structure  $\langle \sqsubseteq, \perp, \sqcup, \sqcap, \top \rangle$ , such that  $\sqsubseteq \equiv \subseteq$ ,  $\sqcup \equiv \cup$ , and  $\sqcap \equiv \cap$  are conveniently lifted to consider the element  $\perp$ .

We say  $\sigma_\ell \in \mathcal{D}$  is the approximate set of variables at location  $\ell$  that may affect the slicing criterion. We will abuse notation to denote the dependencies associated to a symbolic state  $\nu$  also as  $\sigma_\nu$ . *Backward data dependencies* can be formulated using this set, defining two kinds of dataflow information. Given a transition relation  $\ell \xrightarrow{\text{op}} \ell'$  we define  $\text{def}(\text{op})$  and  $\text{use}(\text{op})$  as the sets of variables altered and used during the execution of  $\text{op}$ , respectively. Then,

$$\sigma_\ell \triangleq \begin{cases} (\sigma_{\ell'} \setminus \text{def}(\text{op})) \cup \text{use}(\text{op}) & \text{if } \sigma_{\ell'} \cap \text{def}(\text{op}) \neq \emptyset \\ \sigma_{\ell'} & \text{otherwise} \end{cases} \quad (3.1)$$

where  $\sigma_{\ell'} = V$  if  $\ell' = \ell_{\text{end}}$ . We say a transition relation  $\ell \xrightarrow{\text{op}} \ell'$  where  $\text{op} \equiv x = e$  is included in the slice if:

$$\sigma_{\ell'} \cap \text{def}(\text{op}) \neq \emptyset \quad (3.2)$$

<sup>5</sup>W.l.o.g., we assume in this chapter a single slicing criterion at  $\ell_{\text{end}}$ . A slicing criterion at  $\ell$  can be converted to one at  $\ell_{\text{end}}$  by assigning a special variable  $w$  at  $\ell$  and slicing on  $w$  at  $\ell_{\text{end}}$ .

*Backward control dependencies* can also affect the slicing criterion. A transition relation  $\delta \equiv \ell \xrightarrow{\text{op}} \ell'$  where  $\text{op} \equiv \text{assume}(c)$  is included in the slice if any transition relation under the range of influence<sup>6</sup> (the function  $\text{INFL}$  will compute the range of influence) of  $\delta$  is included in the slice, and

(3.3)

$$\sigma_\ell \triangleq \sigma_{\ell'} \cup \text{use}(\text{op}) \quad (3.4)$$

Finally, a function  $\widehat{\text{pre}}_{\mathcal{D}}(\sigma_\ell, \text{op})$  that returns the *pre-state* after executing backwards the operation  $\text{op}$  with the *post-state*  $\sigma_\ell$  is defined using Eqs. (3.1,3.2,3.3,3.4).

### 3.3 Algorithm

A path-sensitive slicing algorithm over a symbolic execution tree (SET) can be defined as an *annotation* process which labels each symbolic state  $\nu \equiv \langle \ell, \cdot, \cdot \rangle$  with  $\sigma_\ell \in \mathcal{D}$  by computing a fixpoint (later formalised) over the tree, using Eqs. (3.1,3.4) described in Sec. 3.2. In an interleaved process, the final SET is obtained through Eqs. (3.2,3.3). Since the SET may have multiple instances of the same transition relation, we say that a transition relation is included in the final slice if at least one of its instances is included in the slice on the SET. It is easy to see that the path-sensitiveness comes from how symbolic execution builds the tree since no dependencies from a non-executable path can be considered.

Our algorithm performs symbolic execution in a depth-first search manner excluding all infeasible paths. Whenever the forward traversal of a path finishes due to a (a) terminal state, (b) infeasible state, or (c) reusing state (i.e., a state reusing a solution from another state), the algorithm halts and backtracks to the next path. During this backtracking each symbolic state  $\nu$  is labelled with its *solution*, i.e., the set of variables  $\sigma_\nu$  at  $\nu$  that may affect the slicing criterion. Furthermore, the reusing conditions are computed at each state for future use.

We first introduce formally the two key concepts which will decide whether a solution can be reused or not. The first reusing condition is the interpolant, as defined in Definition 1 of Chapter 2. The second reusing condition is the notion

---

<sup>6</sup>More formally, the range of influence for  $\delta$  is the set of transition relations defined in any path from  $\delta$  to its *nearest postdominator* in the transition system.

-  $\sqcup : \mathcal{D}^\omega \times \mathcal{D}^\omega \rightarrow \mathcal{D}^\omega$

$$\sigma^{\omega_1} \sqcup \sigma^{\omega_2} \triangleq \sigma^{\omega_1} \cup \sigma^{\omega_2}$$

-  $\sqsubseteq : \mathcal{D}^\omega \times \mathcal{D}^\omega \rightarrow \text{Bool}$

$$\sigma^{\omega_1} \sqsubseteq \sigma^{\omega_2} \text{ if and only if } \sigma^{\omega_1} \subseteq \sigma^{\omega_2}$$

-  $\widehat{pre} : \mathcal{D}^\omega \times (\Sigma \times \Sigma \times \text{Ops}) \times (\text{Vars} \rightarrow \text{SymVars}) \rightarrow \mathcal{D}^\omega$ .

$$\widehat{pre}(\sigma^{\omega'}, \ell \xrightarrow{\text{op}} \ell', s) \triangleq \left\{ \begin{array}{l} \text{let } \sigma^\omega = \widehat{pre\_aux}(\sigma^{\omega'}, \ell \xrightarrow{\text{op}} \ell', s) \\ \text{foreach } \langle x, \omega_x \rangle \in \sigma^\omega, \langle x, \omega_{x'} \rangle \in \sigma^{\omega'} \\ \quad \sigma^\omega = \sigma^\omega \setminus \{ \langle x, \omega_x \rangle, \langle x, \omega_{x'} \rangle \} \\ \quad \text{if } \omega_x \models \omega_{x'} \text{ then } \sigma^\omega = \sigma^\omega \sqcup \{ \langle x, \omega_{x'} \rangle \} \\ \quad \text{else } \sigma^\omega = \sigma^\omega \sqcup \{ \langle x, \omega_x \rangle \} \\ \quad \text{if } (\sigma^\omega \cap \text{def}(\text{op}) \text{ or } \text{INFL}(\ell \rightarrow \ell') \cap \mathbf{S} \neq \emptyset) \text{ then} \\ \quad \quad \mathbf{S} = \mathbf{S} \cup \{ \ell \rightarrow \ell' \} \\ \text{in } \sigma^\omega \end{array} \right.$$

where:  $\widehat{pre\_aux}(\sigma^{\omega'}, \ell \xrightarrow{\text{op}} \ell', s) \triangleq$

$$\left\{ \begin{array}{l} \{ \langle x, \omega_x \wedge \llbracket y = e \rrbracket_s \rangle \mid \langle x, \omega_x \rangle \in \sigma^{\omega'}, \text{op} \equiv y = e, x \notin \text{def}(\text{op}) \} \cup \\ \{ \langle v, \omega_x \wedge \llbracket y = e \rrbracket_s \rangle \mid \langle x, \omega_x \rangle \in \sigma^{\omega'}, \text{op} \equiv y = e, x \in \text{def}(\text{op}), v \in \text{use}(\text{op}) \} \cup \\ \{ \langle x, \omega_x \wedge \llbracket c \rrbracket_s \rangle \mid \langle x, \omega_x \rangle \in \sigma^{\omega'}, \text{op} \equiv \text{assume}(c) \} \cup \\ \{ \langle x, \llbracket \Pi \pi \rrbracket_s \wedge \llbracket c \rrbracket_s \rangle \mid \langle x, \cdot \rangle \notin \sigma^{\omega'}, \text{op} \equiv \text{assume}(c), x \in \text{use}(\text{op}), \\ \quad \text{INFL}(\ell \rightarrow \ell') \cap \mathbf{S} \neq \emptyset, \exists \pi \equiv \ell' \cdot \dots \cdot \ell_{end} \} \end{array} \right.$$

Figure 3.3: Main Abstract Operations for  $\mathcal{D}^\omega$

of witness paths, as defined in Definition 2, which is slightly tweaked here to consider the slicing criterion and dependency variables.

**Definition 5** (Witness Paths for a Dependency Variable). *Given a symbolic state  $\mathfrak{v} \equiv \langle \ell, \cdot, \cdot \rangle$  annotated with the set of variables  $\sigma_{\mathfrak{v}}$  that affect the slicing criterion at  $\ell_{end}$ , a witness path for a variable  $v \in \sigma_{\mathfrak{v}}$  is a symbolic path  $\pi \equiv \langle \ell, \cdot, \cdot \rangle \cdot \dots \cdot \langle \ell_{end}, \cdot, \Pi_{end} \rangle$  with the final symbolic state  $\mathfrak{v}' \equiv \langle \ell_{end}, \cdot, \Pi_{end} \rangle$  such that  $\llbracket \mathfrak{v}' \rrbracket$  is satisfiable (i.e.,  $\pi$  is feasible). We call  $\llbracket \mathfrak{v}' \rrbracket$  the witness formula of  $v$ , denoted  $\omega_v$ .*

Intuitively, a witness path for a variable at a node is a path below the node along which the variable affects the slicing criterion at the end. A witness formula represents a condition sufficient for the variable to affect the slicing criterion along the witness path.

Prior to establishing the reusing conditions, we augment the abstract domain  $\mathcal{D}$  to accommodate the witness formulas. Here, and in the rest of the chapter, we will refer to the term “dependency” as the set of variables that may affect the

slicing criterion together with their witnesses.

**Definition 6** ( $\mathcal{D}^\omega$ ). We define a new abstract domain  $\mathcal{D}^\omega$  as a lattice  $\langle \sqsubseteq, \perp, \sqcup, \top \rangle$  such that  $\mathcal{D}^\omega \triangleq \{\perp\} \cup \mathcal{P}(\text{Vars} \times \text{FOL})$  (i.e., set of pairs of the form  $\langle x, \omega_x \rangle$  where  $x$  is a variable and  $\omega_x$  is its witness formula) and abstract operations described in Fig. 3.3.<sup>7</sup>

Note that the witness formulas can be obtained only from (feasible) paths in the program. Therefore, the number of witness formulas is always finite. As we will see later, even with loops, the size of each witness formula is also finite because we make the symbolic subtree of the loop finite. That is, we perform symbolic execution on a finite program once loop invariants are given. This ensures that the abstract domain  $\mathcal{D}^\omega$  is finite and hence, termination is guaranteed for any fixpoint computation based on it.

In Fig. 3.3, the operator  $\sqcup$  computes the least upper bound of the abstract states by simply applying the set union of the two set of states. The operator  $\sqsubseteq$  is simply defined to be the subset relation.  $\widehat{pre}$  is a bit more elaborated but basically consists of the Eqs. (3.1,3.2,3.3,3.4) defined in Sec. 3.2 extended with witnesses formulas. We assume here and in the algorithm in Fig. 3.4 that  $\widehat{pre}$  accesses  $\mathbf{S}$  which is the set of transitions included in the slice. In function  $\widehat{pre\_aux}$ , there are four cases to handle different kinds of statements and dependencies:

- In the first two cases, if the operation is an assignment, the dependencies are propagated from the *defined* to the *used* variables and any dependency from a variable not *defined* is kept. In these cases, the pre-state witness formula is the conjunction of the post-state witness formula with the corresponding statement.
- In the third case, if the operation is an assume, any *used* variable is preserved, with its pre-state witness formula being the conjunction of the post-state witness formula and the corresponding guard.
- In the last case, for any variable  $x$  occurring in an assume statement without any dependency, if any transition under the range of influence (computed by INFL) of the assume is already in the slice, then  $x$  is added (due to control dependency) and its witness formula is the conjunction of the

---

<sup>7</sup>For clarity, trivial treatment of the element  $\perp$  is omitted from operations in Fig. 3.3.

guard and the path condition of any (feasible) path from the assume statement that leads to the end of the program.

In addition, in function  $\widehat{pre}$  whenever two pairs from the set of dependencies computed by  $\widehat{pre\_aux}$  refer to the same variable, we choose the one with the weaker witness formula (which is more likely to be reused) by checking if one logically implies the other. Finally, a transition is included in the slice if one of the Eqs. (3.2,3.3) holds.

We now define our main condition for reusing dependencies computed at another node, very similar to the merging conditions in Definition 3.

**Definition 7** (Reusing Conditions). *Given a current symbolic state  $\nu \equiv \langle \ell, \cdot, \Pi \rangle$  and an already solved symbolic state  $\nu' \equiv \langle \ell, \cdot, \cdot \rangle$  such that  $\bar{\Psi}$  is the interpolant generated for  $\nu'$  and  $\sigma^\omega$  are the dependencies together with their attached witnesses at  $\nu'$ , we say  $\nu$  is equivalent to  $\nu'$  (or  $\nu$  can reuse the solution at  $\nu'$ ) if the following conditions hold:*

$$\begin{aligned} (a) \quad \llbracket \nu \rrbracket &\models \bar{\Psi} \\ (b) \quad \forall \langle x, \cdot \rangle \in \sigma^\omega &\bullet \exists \langle x, \omega_x \rangle \in \sigma^\omega \text{ such that } \llbracket \nu \rrbracket \wedge \omega_x \text{ is satisfiable} \end{aligned} \tag{3.5}$$

The condition (a) affects *soundness* and it ensures that the set of symbolic paths reachable from  $\nu$  must be a subset of those from  $\nu'$ . The condition (b) is the witness check which essentially states that for each variable  $x$  in the dependency set at  $\nu'$ , there must be at least one witness path with formula  $\omega_x$  that is feasible from  $\nu$ . This affects *accuracy* and ensures that the reuse of dependencies does not incur any loss of precision.

We now describe in detail the main features of our algorithm defined by the function  $\text{BackwardDeps}_V$  in Fig. 3.4. The main purpose of  $\text{BackwardDeps}_V$  is to keep track of the *backward dependencies* between the program variables and the slicing criterion by inferring for each state the set of variables that may affect the slicing criterion. From these dependencies it is straightforward to obtain the slice of the program as explained at the beginning of this section. For clarity of presentation, let us omit the content of the *grey* boxes and assume programs do not have loops, which we will come to later.

$\text{BackwardDeps}_V : \text{SymStates} \times \mathcal{D}^\omega \rightarrow \text{FOL} \times \mathcal{D}^\omega \times \text{Bool}$  requires the pro-



BackwardDeps<sub>V</sub>( $\nu \equiv \langle \ell, s, \Pi \rangle, \sigma^\omega$ )

- 1:  $\text{change} = \text{false}$
- 2: **if** INFEASIBLE( $\nu$ ) **then**  $\langle \bar{\Psi}, \sigma^\omega \rangle = \langle \text{false}, \emptyset \rangle$  and **goto** 13
- 3: **if** TERMINAL( $\nu$ ) **then**  $\langle \bar{\Psi}, \sigma^\omega \rangle = \langle \text{true}, \{ \langle \nu, \text{true} \rangle \mid \nu \in V \} \rangle$  and **goto** 13
- 4: **if**  $\exists \nu' \equiv \langle \ell, s, \cdot \rangle$  labelled with  $\langle \bar{\Psi}, \sigma^\omega \rangle$  such that REUSE( $\nu, \nu'$ ) **then goto** 13

- 5: **if**  $\ell$  is the header of a loop **then**
- 6:    $\bar{\nu} = \text{invariant}(\nu, \ell \rightarrow \dots \rightarrow \ell)$
- 7:    $\langle \bar{\Psi}, \sigma^\omega, \text{change} \rangle = \text{UnwindTree}_V(\bar{\nu}, \sigma^\omega)$  and **goto** 13
- 8:   **if**  $\exists \ell'$  such that  $\ell \rightarrow \ell'$  is a backedge of a loop **then**
- 9:     $\langle \cdot, \cdot, \bar{\Pi} \rangle = \text{invariant}(\nu, \ell' \rightarrow \dots \rightarrow \ell)$
- 10:    $\langle \bar{\Psi}, \sigma^\omega \rangle = \langle \bar{\Pi}, \sigma^\omega \rangle$  and **goto** 13

- 11:  $\langle \bar{\Psi}, \sigma^\omega, \text{change} \rangle = \text{UnwindTree}_V(\nu, \sigma^\omega)$
- 12:  $\text{change} = \text{change} \vee \nu$  is labelled with  $\langle \cdot, \sigma^{\omega_{old}} \rangle$  such that  $\neg(\sigma^{\omega_{old}} \sqsubseteq_{\mathcal{D}^\omega} \sigma^\omega)$
- 13: label  $\nu$  with  $\langle \bar{\Psi}, \sigma^\omega \rangle$  and **return**  $\langle \bar{\Psi}, \sigma^\omega, \text{change} \rangle$

UnwindTree<sub>V</sub>( $\nu \equiv \langle \ell, s, \Pi \rangle, \sigma^{\omega_{in}}$ )

- 1:  $\bar{\Psi} = \text{true}, \sigma^\omega = \sigma^{\omega_{in}}, \text{change} = \text{false}$
- 2: **foreach** transition relation  $\ell \xrightarrow{\text{op}} \ell'$
- 3:
 
$$\nu' \triangleq \begin{cases} \langle \ell', s, \Pi \wedge \llbracket c \rrbracket_s \rangle & \text{if op} \equiv \text{assume}(c) \\ \langle \ell', s[x \mapsto S_x], \Pi \wedge \llbracket x = e \rrbracket_s \rangle & \text{if op} \equiv x = e \text{ and } S_x \text{ fresh variable} \end{cases}$$
- 4:  $\langle \bar{\Psi}', \sigma^{\omega'}, c \rangle = \text{BackwardDeps}_V(\nu', \sigma^{\omega_{in}})$
- 5:  $\bar{\Psi} = \bar{\Psi} \wedge \widehat{wlp}(\text{op}, \bar{\Psi}')$
- 6:  $\sigma^\omega = \sigma^\omega \sqcup_{\mathcal{D}^\omega} \widehat{pre}_{\mathcal{D}^\omega}(\sigma^{\omega'}, \text{op}, s)$
- 7:  $\text{change} = \text{change} \vee c$
- 8: **return**  $\langle \bar{\Psi}, \sigma^\omega, \text{change} \rangle$

BackwardDepsLoop<sub>V</sub>( $\nu, \sigma^\omega$ )

- 1:  $\sigma^{\omega'} = \sigma^\omega, \text{change} = \text{false}$
- 2: **do**  $\langle \cdot, \sigma^{\omega'}, \text{change} \rangle = \text{BackwardDeps}_V(\nu, \sigma^{\omega'})$  **while**  $\text{change}$  **end**

Figure 3.4: Path-Sensitive Backward Slicing Analysis

gram to have been translated to a transition system  $\langle \Sigma, I, \longrightarrow, O \rangle$  and takes as input an initial symbolic state  $\nu \equiv \langle \ell \in I, \varepsilon, \text{true} \rangle$  and an initially empty  $\sigma^\omega$ .  $V$  is the set of variables of the slicing criterion. The set of transitions included in the slice,  $\mathbf{S}$ , is also empty. Recall that  $\mathbf{S}$  is only modified by  $\widehat{pre}$ , and hence, we omit it from the description of the algorithm defining it as a global variable. The output is a triple with the interpolant, dependencies (i.e., reusing conditions and solution) and a boolean flag representing whether any change occurred in a dependency set at any symbolic state during the algorithm's backward traversal

(this is used mainly to handle loops later). The actual object of interest computed by the algorithm is the set of transitions  $\mathbf{S}$  included in the slice.

$\text{BackwardDeps}_V$  implements a recursive algorithm whose objective is to generate a finite complete SET while reusing solutions whenever possible to avoid path explosion. Line 1 initialises the (local) variable  $\text{change}$  to *false*, which will be updated later. Next, the three base cases for symbolic states are handled - infeasible, terminal, and reuse:

- In line 2, the function  $\text{INFEASIBLE}(\langle \cdot, \cdot, \Pi \rangle)$  checks whether  $\Pi$  is satisfiable. If not, the symbolic execution detects an infeasible path and halts, excluding any dependency which would have been inferred from the non-executable path. In addition, it produces an interpolant from  $\Pi$  and *false*, namely  $\bar{\Psi} \equiv \text{false}$ , which generalises the current path condition ( $\Pi \models \bar{\Psi}$  and  $\bar{\Psi}$  is *false*). Since the path is not executable there is no variable that may affect the slicing criterion and hence, the set of dependencies returned is empty.
- In line 3, the function  $\text{TERMINAL}(\langle \ell, \cdot, \cdot \rangle)$  checks if the symbolic state is a terminal node by checking if  $\ell = \ell_{\text{end}}$ . If yes, the execution has reached the end of a path. Since the path is feasible, it can be fully generalised returning the interpolant  $\bar{\Psi} \equiv \text{true}$ . Since  $\ell$  is a terminal node, the set of dependencies is the set of variables in the slicing criterion,  $V$ . The witness formula for each variable from  $V$  is initially *true*.
- In line 4 the algorithm searches for another state  $\nu'$  whose dependencies can be reused by the current state  $\nu$  so that the symbolic execution can be stopped. For this, the function  $\text{REUSE}(\nu, \nu')$  tests both the reusing conditions in Eq. 3.5. If the test holds, the state  $\nu$  can reuse the dependencies computed by  $\nu'$ .

If all three base cases fail, the algorithm unwinds the execution tree by calling the procedure  $\text{UnwindTree}_V$  at line 11.  $\text{UnwindTree}_V$ , at line 3, executes one symbolic step<sup>8</sup> and calls the main procedure  $\text{BackwardDeps}_V$  with the successor state (line 4). After the call the two key remaining steps are to compute:

---

<sup>8</sup>Note that the rule described in line 3 is slightly different from the one described in Sec. 3.2 because no consistency check is performed. Instead, the consistency check is postponed and done by the first base case at line 2.

- the interpolant  $\bar{\Psi}$  (UnwindTree<sub>V</sub> line 5) that generalises the symbolic execution tree below  $\mathfrak{v}$  while preserving its infeasible paths. The procedure  $\widehat{wlp} : Ops \times FOL \rightarrow FOL$  ideally computes the *weakest liberal precondition (wlp)* [36] which is the weakest formula on the initial state ensuring the execution of  $op$  results in a final state  $\bar{\Psi}'$ . In practice, we approximate  $wlp$  by making a linear number of calls to a theorem prover following techniques described in [66]. The interpolant  $\bar{\Psi}$  is an FOL formula consisting of the conjunction of the result of  $\widehat{wlp}$  on each child's interpolant.
- the solution,  $\sigma^\omega$ , for the current state  $\mathfrak{v}$  at line 6 which is computed by executing  $\widehat{pre}_{\mathcal{D}^\omega}$  on each child's solution and then combining all solutions using  $\sqcup_{\mathcal{D}^\omega}$ .

In addition, at line 7 it also records changes in any child's symbolic state (if any) and then returns a triple in the same format as BackwardDeps<sub>V</sub>'s return value. In BackwardDeps<sub>V</sub>, line 12 updates *change* to *true* if either it was set to *true* in UnwindTree<sub>V</sub> at line 11 or the current symbolic state is about to be updated with a more precise solution than that it already has. The final operation before returning from BackwardDeps<sub>V</sub> is to label the state  $\mathfrak{v}$  with the reusing conditions and solution (line 13).

Now we continue describing our algorithm by discussing how it handles loops. The main issue is to produce a finite symbolic execution tree on which a fixpoint of the dependencies can be computed.

For this, the algorithm in Fig. 3.4 takes an annotated transition system in which program points are labelled with inductive invariants inferred automatically by an abstract interpreter using an abstract domain such as *octagons* or *polyhedra* (we borrow the ideas presented in [98] for this purpose). We assume the abstract interpreter provides a function `getAssrt` which, given a program location  $\ell$  and a symbolic store  $s$ , returns an assertion in the form of an FOL formula renamed using  $s$ , which holds at  $\ell$ . Note that when applied at loop headers, `getAssrt` will return a loop invariant. However, we would like to strengthen it using the constraints propagated from the symbolic execution. The function `invariant` performs this task as follows:

$$\text{invariant}(\langle \ell, s, \Pi \rangle, \ell_1 \rightarrow \ell_n) \triangleq \begin{cases} \mathbf{let} \ s' = \text{havoc}(s, \text{modifies}(\ell_1 \rightarrow \ell_n)) \\ \quad \bar{\Pi} = \text{getAssrt}(\ell, s') \wedge \Pi \\ \mathbf{in} \ \langle \ell, s', \bar{\Pi} \rangle \end{cases}$$

$\text{havoc}(s, \text{Vars}) \triangleq \forall v \in \text{Vars} \bullet s[v \mapsto z]$

where  $z$  is a fresh variable (implicitly  $\exists$ -quantified).

$\text{modifies}(\ell_1 \rightarrow \dots \rightarrow \ell_n)$  takes a sequence of transitions and returns the set of variables that may be modified during its symbolic execution.

Intuitively, `invariant` clears the symbolic store of all variables modified in the loop (using the `havoc` function) and then enhances the path condition  $\Pi$  of the symbolic state with the invariants from the abstract interpreter.

Let us now explain the *grey* boxes in Fig. 3.4. Lines 5-7 in `BackwardDepsV` cover the case when a loop header has been encountered. The main purpose here is to abstract the current symbolic state by using the loop invariant obtained from the abstract interpreter. The algorithm calls the function `invariant` (at line 6) with the transitions in the loop so as to obtain a copy of the current symbolic state annotated with the approximate loop invariant in its path condition. At line 7, the `UnwindTreeV` procedure is called on the resulting symbolic state to explore the symbolic subtree of the loop.

If the symbolic execution encounters a loop backedge (lines 8-10) from  $\ell$  to  $\ell'$  it halts and backtracks. The reason is that the loop header at  $\ell'$  has already been symbolically executed with a loop invariant. Hence there is no need to continue the loop since the invariant ensures that no new feasible paths will be encountered if it is explored again. This is our basic mechanism to make the symbolic execution of the loop finite.

Finally, the main algorithm to handle loops, `BackwardDepsLoopV`, makes calls to the function `BackwardDepsV` until there is no change detected in the symbolic state of any program point. We present it in its simplest form, but it can be easily optimised to call `BackwardDepsV` only with the loop in which the change was detected.

## 3.4 Experimental Evaluation

We implemented the path-sensitive slicer described in this chapter and performed experiments to address the following questions:

1. Is our path-sensitive slicer practical for medium-size programs?
2. What is the impact of *reusing* ?
3. How effective is a path-sensitive slicer against a path-insensitive version?

Our proof-of-concept implementation models the heap as an array. A flow-insensitive pointer analysis is used to partition updates and reads into alias classes where each class is modelled by a different array. Given an operation that involves pointers the sets *def* and *use* utilise the results of the pointer analysis. For instance, given the statement  $*p = *q$  the set *def* contains everything that might be pointed to by  $p$  and the set *use* includes everything that might be pointed to by  $q$ . A theorem prover is used to decide linear arithmetic formulas over integer variables and array elements in order to check the satisfiability and entailment of formulas, and computing interpolants and witnesses. Programs are first annotated with approximate loop invariants using the abstract interpreter InterProc [74]. Functions are inlined (hence recursion is not supported) and external functions are modelled as having no side effects and returning an unknown value.

We used several instrumented device driver programs previously used as software model checking benchmarks: *cdaudio*, *diskperf*, *floppy*, and *serial*. In addition, we also considered *mpeg*, the *mpeg-1* algorithm for compressing video, and *fcron.2.9.5*, a cron daemon. For the slicing criterion we consider variables that may be of interest during debugging tasks. For the instrumented software model checking programs, we choose as the slicing criterion the set of variables that appear in the safety conditions used for their verification in [53]. In the case of *mpeg* we choose a variable that contains the type of the video to be compressed. Finally, in *fcron.2.9.5* we choose all the file descriptors opened and closed by the application.

Table 3.1 compares our path-sensitive slicer (columns labelled with Path-Sens) against the same slicer but without path-sensitivity (labelled with Path-Insens). Path-insensitivity is achieved by the following modifications in our

Program	LOC	Path-Insens		Path-Sens		
		Size Red	Time	Reuse		No Reuse
				Size Red	Time	Time
mpeg	5K	4%	21s	8%	628s	$\infty^1$
diskperf	6K	32%	2s	57%	94s	$\infty$
floppy	8K	36%	9s	47%	263s	$\infty$
cdaudio	9K	23%	10s	52%	301s	$\infty$
serial	12K	39%	16s	50%	395s	$\infty$
fcron.2.9.5	12K	42%	32s	61%	832s	$\infty$
Mean		<b>23%</b>	<b>15s</b>	<b>38%</b>	<b>418s</b>	—

Table 3.1: Results on Intel 3.2Gz 2Gb. <sup>1</sup> timeout after 2 hours or 2.5 Gb of memory consumption

slicer: (1) considering all paths as feasible, and (2) always forcing reuse. These changes have the same effect as always merging the abstract states along incoming edges in a control-flow merging node. In other words, they mimic running a path-insensitive slicer on the original CFG. We could have used a faster off-the-shelf path-insensitive program slicer (using e.g., [57]), however, our objective here is to isolate the impact of path-sensitivity and hence, we decided to perform the comparison on a common platform to produce the fairest results. Finally, we also tried running with different abstract domains, such as octagons and polyhedra, to generate loop invariants and the results were the same.

The column LOC represents the number of lines of program without comments. The column Size Red shows the reduction in slice size (in %) w.r.t. the original program size. The reduction size is computed using the formula  $(1 - \frac{\text{size of slice}}{\text{size of original}}) \times 100$ . By *size* we mean all executable statements in the program, excluding type declarations, unused functions, comments, and blank lines. A minor complication here is that the SET may contain multiple *instances* of program points in the CFG, as can be seen in Fig. 3.1(c). To compare the reduction in slice sizes fairly, we use the rule mentioned at the beginning of Sec. 3.3 to compute slices: a transition in the original CFG is included in the slice if any of its instances in the SET is included in the slice.

The column Time reflects the running time of the analysis in seconds excluding the alias analysis and the external abstract interpreter. Column Reuse is our path-sensitive slicer with reusing, and No Reuse uses the same symbolic execution engine with automatic loop invariants but without interpolation and witness

paths. Finally, we summarise in row Mean the numbers of columns Size Red and Time by computing their *geometric* and *arithmetic* mean, respectively.

We summarise our results as follows. The running times (column Reuse) of our path-sensitive slicer (with a mean of 418 secs) are reasonable considering the size of the programs and the current status of our prototype implementation which can be optimised significantly. The analysis of `mpeg` is especially slow and it is due to the existence of many nested loops. On the other hand, the reuse of solutions clearly pays off. Without our reuse mechanism (column No Reuse) we were not able to finish the analysis of any program after a timeout of 2 hours or memory consumption of 2.5 Gb. Finally, the improvement in terms of reduction shown in column Reuse is roughly 38% against only 23% of its path-insensitive counterpart (column Path-Insens). Again, the `mpeg` program is an exception since the size of the slices in both Path-Insens and Path-Sens are quite big (i.e., very small reduction). The reason is that in `mpeg` all the computations depend on the type of video to be compressed which is our slicing criterion.

### 3.5 Related Work

Static slicing remains a very active area of research. We limit our discussion to the most relevant works that take into account path-sensitiveness. We also discuss pruning techniques that might have influenced our work.

**Fully path-sensitive methods.** Conditioned slicing [19, 31, 33] performs symbolic execution in order to exclude infeasible paths before applying a static slicing algorithm, so they are fully path-sensitive (for loop-free programs) similar to us. However, they perform full path enumeration and essentially explore the search space of the naive SET. Hence, they suffer from path explosion.

**Partially path-sensitive methods.** A more scalable but not fully path-sensitive approach is described by Snelting et al. [100, 90, 99]. They compute the dependency between two program points  $y$  and  $x$  using the Program Dependence Graph (PDG) [57] and apply the following rule to remove spurious dependencies:  $I(y, x) \Rightarrow \exists \bar{v} : PC(y, x)$ , where  $I(y, x)$  stands for  $y$  influences  $x$  (i.e., there is a dependency at  $x$  on  $y$ ),  $\bar{v}$  is some assignment of values to program variables

and  $PC(y,x)$  is the path condition from  $y$  to  $x$ . Essentially it means that if the path condition from  $y$  to  $x$  is found to be unsatisfiable, then there is definitely no influence from  $y$  to  $x$ . If there are multiple paths between two points, the path condition is computed as a disjunction of each path.

For the program in Fig. 3.1(a), [100, 90, 99] would proceed as follows. In the PDG there will be a dependency edge from  $\ell_8$  to  $\ell_1$ , hence they would check to see if the path condition  $PC(1, 8)$  is unsatisfiable. First they calculate the path condition from  $\ell_4$  to  $\ell_8$  as  $PC(4, 8) \equiv (x = 1 \wedge y > 0 \wedge z = x) \vee (y = 0 \wedge y > 0 \wedge z = x) \equiv (x = 1 \wedge y > 0 \wedge z = x)$ . Now they use this to calculate  $PC(1, 8) \equiv (x = 0 \wedge y = 5 \wedge ((a > 0 \wedge b = x + y \wedge PC(4, 8)) \vee (a \leq 0 \wedge PC(4, 8))))^9$ , which is not unsatisfiable. Hence the statement  $x=0$  at  $\ell_1$  will be included in the slice.

The fundamental reason for this is that for [100, 90, 99], path conditions are only necessary and not sufficient, so false alarms in examples such as the above are possible. An important consequence of this is the fact that even for loop-free programs, their algorithm cannot be considered “exact” in the sense described in Sec. 3. However, our algorithm guarantees to produce no false alarms for such programs.

Finally, another slicer that takes into account path-sensitiveness up to some degree is Constrained slicing [43] which uses *graph rewriting* as the underlying technique. As the graph is rewritten, modified terms are tracked. As a result, terms in the final graph can be tracked back to terms in the original graph identifying the slice of the original graph that produced the particular term in the final graph. The rules described in [43] mainly perform constant propagation and dead code detection but not systematic detection of infeasible paths. More importantly, [43] does not define rules to prune the search space.

**Interpolation and SAT.** Interpolation has been used in software verification [9, 53, 82, 59] as a technique to eliminate facts which are irrelevant to the proof. Similarly, SAT can explain and record failures in order to perform conflict analysis. By traversing a reverse implication graph it can build a *no-good* or conflict clause which will avoid making the same wrong decision. Our

---

<sup>9</sup>We have simplified this formula since [100, 90, 99] uses the SSA form of the program and adds constraints for  $\Phi$ -functions, but the essential idea is the same.



algorithm has in common the use of some form of *nogood learning* in order to prune the search space. But this is where the similarity ends. A fundamental distinction is that in program verification there is no solution (e.g., backward dependencies) to compute and hence, there is no notion of reuse and the concept of witness paths does not exist. [44] uses interpolation-based model checking techniques to improve the precision of dataflow analysis but still for the purpose of proving a safety property.

Finally, the recent work [67] has been a clear inspiration for this work. [67] uses interpolation and witnesses as well to solve not an analysis problem, but rather, a *combinatorial optimisation* problem: the Resource-Constrained Shortest Path (RCSP) problem. Moreover, there are other significant differences. First, [67] is totally defined in a finite setting. Second, [67] considers only the narrower problem of extraction of bounds of variables for loop-free programs while we present here a general-purpose program analysis like slicing. Third, this work presents an implementation and demonstrates its practicality on real programs.

### 3.6 Summary

We have so far presented a path-sensitive backward slicer. The main result is a symbolic execution based algorithm which excludes infeasible paths while pruning redundant paths. The key idea is to halt the symbolic execution while reusing dependencies from other paths if some conditions hold. The conditions are based on a notion of interpolation and witness paths aiming to detect whether the exploration of a path can improve the accuracy of the dependencies computed so far by other paths. We have demonstrated the practicality of the approach with a set of real C programs.

Finally, we want to mention that although this work targets slicing our approach can in fact be generalised and applied to other backward program analyses providing them path-sensitiveness. Nevertheless, note that there are other factors that may affect the effectiveness of our approach: precision of the numerical abstract domain to generate the loop invariants, completeness of the theorem prover (e.g., no support for bitwise operations), etc. But these issues are orthogonal to our main contribution.

## Part II: Slice-based Program Transformation

Based on our previous result that slicing is more effective when there is path-sensitivity, we extend our slicing to a more general *program transformation* method. Specifically, the transformation will produce a new intermediate graph representation for C programs with specified target variables. These programs are intended to be processed by third-party applications such as verifiers and testers. The representation embodies two concepts. First, it is *path-sensitive* in the sense that there may be multiple nodes representing one program point so that infeasible symbolic execution paths can be excluded. Second, and more importantly, the graph is *sliced* with respect to the target variables.

We begin with a promotional example: consider the C program

```
if (c) p = 1; else p = 0;
x = 0;
if (p > 0) x = 1;
if (x == 0) z = 1;
TARGET: {z}
```

No static slicing is effective on this program because each statement and variable affects the target  $z$  along at least one path. However, we can *transform* this program into an equivalent one:

```
if (!c) z = 1;
```

which produces, on any given input, the same values for  $z$  as the original program. Clearly this transformed program would be more efficient when input to a verifier or tester which seeks properties of  $z$ . We arrived at this transformation as follows. Let  $S$  denote the program fragment comprising of all but the first if-statement of the original program. Now consider slicing  $S$  in the *context*  $p = 1$  (the “then” body of the first if-statement). Clearly  $S$  would not modify the variable  $z$  because only the statements  $x=0$  and  $x=1$  would be executed in this context. Next consider the alternative context  $p = 0$  (the “else” body). Now  $S$

would execute the statements  $x=0$  and  $z=1$ , from which the former can be sliced away, as it does not affect the target  $z$ . Hence we get the transformed program.

In other words, we arrived at this new program by first considering path-sensitivity, and more specifically, the original program's *symbolic execution tree*. The general idea is that slicing of a program fragment can be much more effective when it is done with a given context. A symbolic execution tree in fact displays the context of a program fragment as it unfolds through the various paths that bring execution to this fragment. Now consider the example:

```
if (c) p = 1; else p = 0;
S
TARGET: {z}
```

where  $S$  now represents a program fragment which *cannot* be sliced by restricting consideration of the values of  $z$  at the end of the program. That is, all symbolic execution paths in  $S$  produce some (different) output value in  $z$  at the end, regardless of the initial values of  $c$  or  $p$ . Here, by being path-sensitive, we would produce a CFG that corresponds to the program:

```
if (c) { p = 1; S; }
else { p = 0; S; }
```

This new program is effectively twice the size of the original program due to the duplication of  $S$ , and yet there is no benefit from using this enlarged representation.

It is folklore that a fully path-sensitive representation of symbolic execution is simply intractable, for the representation doubles in size for each branch statement encountered. The only alternative is to have some form of *merging* where, at some stages in the construction of the graph, certain paths in the graph transition into the *same* node. If the merging is performed at every opportunity, the original CFG would be obtained. If not performed at all, the full, possibly intractable, symbolic execution tree would be obtained. The big question is, therefore, *how much* merging is needed?

In this work, we present a method for producing a path-sensitive CFG by constructing a symbolic execution tree but merging nodes *exactly when* the

merge does not hide any information that affects slicing. That is, when our algorithm merges a node in the tree with another, it guarantees that had the node been symbolically executed instead, one would obtain precisely the same slicing information as that of the node it's being merged with. A key step involved in the construction of our CFG, which we call the *Path-Sensitively Sliced CFG* or *PSS-CFG* for short, is “Tree Slicing”, a powerful technique to merge and slice arbitrarily different symbolic execution sub-trees under certain conditions.

Our main result is that the PSS-CFG, when “decompiled” (or transformed) into regular programs that can be directly used by applications which query target variables (e.g., a concolic tester that targets the program’s outputs, or a verifier with a safety property), produces significant improvement in terms of time usage as compared to using the original program. The strength of the PSS-CFG is that it can be used “out-of-the-box” by a wide number of third-party software engineering applications. We consider two main applications - program testing and verification, and show in Section 3.11 how they can benefit from the PSS-CFG to gain in performance.

### 3.7 Related Work

With regards to performing an “offline” program transformation for general use, our closest related work is by Balakrishnan *et al.* [8], performing a path-sensitive transformation with an aim to “improve the path-insensitive analysis used inside the F-SOFT tool to obtain the effects of path-sensitive analysis”. The main difference with our work is that their transformation only removes infeasible paths from the CFG without performing slicing. As a result, it is not clear how the performance of a verifier or tester could be improved because they are themselves path-sensitive and hence would not consider the infeasible paths anyway. Hence, their target application is a *path-insensitive* analyser within F-SOFT that can benefit from the removal of infeasible paths, as it would spuriously consider them otherwise. Nevertheless we share with them the high level goal of performing an “offline” program transformation that helps an external application.

Since program transformation is a very specific area, we also discuss re-

lated work that do not perform transformation but still provide benefits of path-sensitivity in general for external consumption. In this regard, another related work is by Boonstoppel *et al.* [15] that discards irrelevant tests in concolic testing by tracking values read and written by executed program statements. Then, when two states differing only in program values *not subsequently read* are visited, the exploration of the second state can be pruned. The main difference is that they use live range information of variables to make the decision about pruning paths, which results in lesser pruning compared to our method that uses slicing (specifically, dependency information). Moreover they do not perform program transformation themselves, but work with the concolic tester to discard certain tests during execution. Thus, they are dependent on the external application, whereas we perform an offline transformation that is independent of the application.

Another related work is by Jhala & Majumdar [69] that performs slicing of paths, but their goal is to reduce the size of the counterexample path generated by CEGAR-based verifiers during their process. As a result, they do not work with the entire program’s CFG and hence, they are not concerned with splits and merges in the CFG. However a program transformation algorithm like ours needs to work with the whole CFG to decide where to split or merge.

The recent work [73] performs dynamic state merging during symbolic execution in order to combine paths. While reducing the number of paths, the formulas corresponding to merged states are more complicated. They showed that their chosen heuristic for deciding which states are merged produced significant speedups in overall symbolic execution. A similarity to our work here is that we also perform state merging but we do so by learning when different states need not be explored. Also, [73] does not consider slicing, but our algorithm merges only when it can guarantee lossless-ness of slicing (dependency) information.

We will see in Section 3.10 that our method uses the concept of *interpolation* to perform state merging. Interpolation has been successfully used in program testing and verification [80, 66, 83, 63] to reduce state space and contain “path-explosion”. However the similarity with our work is only in the use

of interpolation for state merging. Our method has to additionally guarantee that the merging is *lossless*, something that is inapplicable to these works. Also, in Section 3.11, we experimentally evaluate the PSS-CFG using applications of testing and verification, which may be another source of confusion with these works. The fundamental difference is that these works involve directly performing the testing or verification process on the program. We do neither of those – we simply happened to *evaluate* the PSS-CFG on third-party testing and verification tools to show that they benefit in performance (in fact, [80] is one of the verifiers used in Section 3.11). But the PSS-CFG is a much more generic object not limited to just testing and verification.

We finally compare with our own work [65], presented in Part I of this chapter, which performed static slicing. The technical approach there was to first generate a path-sensitive symbolic execution tree which then was used to determine which primitive program statements could be sliced. It performed static slicing on the program, using the symbolic tree to slice a statement that does not affect the target *anywhere* in the tree. In contrast, here we perform slicing *on the tree itself* to transform it into a new tree using the transformation rules. Our new method allows the same program statement to be sliced from one part of the tree but not another, a scenario in which we previously simply could not slice the statement from the program at all. This fundamental difference will be exposed in Section 3.8, where our previous work cannot slice anything since every statement is relevant along some path, whereas now we are able to slice statements depending on the contexts in which they are (ir)relevant.

Moreover, a key technical slicing step of our new method, called “Tree slicing” (elucidated in Section 3.8), involves slicing a compound statement *from a tree*, a problem not relevant to our previous setting: in general, the symbolic execution subtrees rooted at what corresponds to the end of a compound statement may not be identical. A main theoretical result concerning the correctness of the transformation (Theorem 3), is that under certain conditions we can correctly slice away the entire compound statement, and *merge* the following subtrees even though they are different, while still retaining the necessary equivalent behaviour of the original program on the target variables.

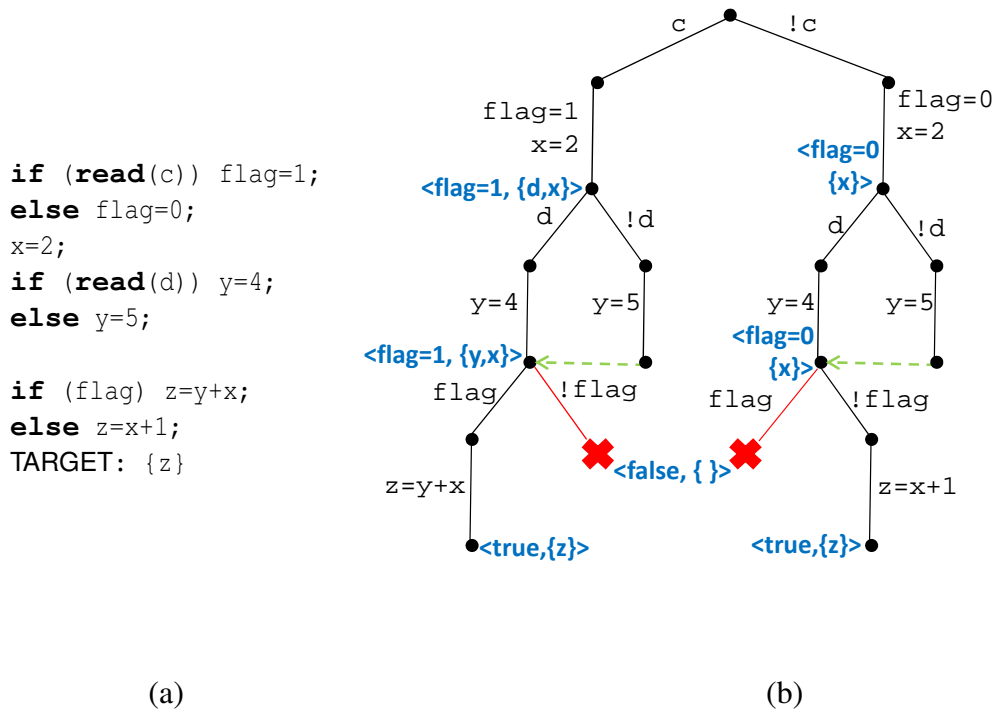


Figure 3.5: A program and its symbolic execution tree

### 3.8 Basic Idea

We now explain our basic idea through an example, using concolic testing to show how it can benefit from the PSS-CFG. Consider the program in Fig. 3.5, where a read call signals the concolic tester to generate an input, and the target of interest (say, the program’s output) is the variable  $z$  in the end. The program has two inputs  $c$  and  $d$  which are used to decide the control flow and there are 8 paths to traverse. At the outset, note that no static slicers, even path-sensitive ones like [65] or the well-known Frama-C [29], are effective on this program because each statement along *some* path affects the target.

Our algorithm has two steps: first, it performs symbolic execution to generate the *symbolic execution tree* (SE tree) that is annotated with dependency information at each node (Fig. 3.5(b)). The goal here is to be as path-sensitive as possible since it makes dependency information more precise. However path-sensitivity also makes the SE tree easily grow to be exponential in the number of branches. Hence the challenge is to keep its size in check by *merging*, but ensuring this does not cause imprecision of dependencies. In the second step, transformation rules are applied on the SE tree to get the final PSS-CFG. These

rules take advantage of the precise dependency information to make different decisions that apply to different contexts of the same program statement. We will now explain both steps in detail.

### Phase One: Building the SE tree

In Fig. 3.5(a), our algorithm first encounters a branch on  $c$ . To be path-sensitive, it splits symbolic execution into two – one with context  $c$  and the other with context  $\neg c$ . In a DFS fashion, it first explores the context  $c$ , symbolically executing the statements  $flag=1$  and  $x=2$  and as usual, carrying these path constraints in a logic formula. Upon reaching the next branch, it again splits into two – with context  $d$  and  $\neg d$ . Continuing along the context  $d$  it executes  $y=4$  and reaches the final branch. Again it splits into two – with context  $flag$  and  $\neg flag$ , following the former and finally executing  $z=y+x$  before reaching the terminal point.

Our algorithm now generates the backwards dependency information for this feasible path, resulting in the dependency set  $\{z\}$ , the target at the terminal point (shown in blue between curly braces). This is then propagated back to the branch point on  $flag$  by applying Weiser’s formulas [104], resulting in the set  $\{y, x\}$ . In addition to the dependency set, our algorithm also computes the *witness path* (see Definition 2) for each variable in the set. For instance, the witness path for the above set of variables is the path executing  $flag$  and  $z=y+x$ , corresponding to the formula  $flag \wedge z = y + x$ .

Now, our algorithm backtracks and explores the other context from the last split point:  $\neg flag$ . The path formula  $c \wedge flag = 1 \wedge x = 2 \wedge d \wedge y = 4 \wedge \neg flag$  is unsatisfiable, hence the path is infeasible. Now it computes the interpolant (see Definition 1)  $flag = 1$  (blue) that captures the essence of infeasibility of the path at the branch point. Thus, at the branch point on  $flag$ , the dependency set is  $\{y, x\}$ , the witness path is  $flag \wedge z = y + x$  and the interpolant is  $flag = 1$ .

Notice that our algorithm did not include the variable  $flag$  in the dependency set, whereas a traditional slicer such as [29, 65] would have included it due to control dependency. The reason is that along this particular path only one branch, the one where  $flag$  is true (non-zero), is feasible and the other is infeasible. Hence the value of  $flag$  does not really affect the execution of



the statement  $z=y+x$ , therefore this statement is not control-dependent on `flag`. However, `flag` being true (non-zero) is needed to preserve the infeasibility of that branch and this is captured in the interpolant. In general, we do not need to add control-dependencies on branch variables where only one branch is feasible.

Next our algorithm backtracks again to the previous split point to explore the other branch: `!d`. It executes the statement  $y=5$  and reaches the branch on `flag`, this time under the different context `!d`. Now, the important step of checking whether the current context of the branch can be merged with the previously explored context is performed, using the merging conditions in Definition 3. Both checks succeed here as (a) the current path formula  $c \wedge flag = 1 \wedge x = 2 \wedge \neg d \wedge y = 5$  implies the interpolant  $flag = 1$ , and (b) the witness path  $flag \wedge z = y + x$  is feasible in the current context. Therefore, the current context of the branch on `flag` can be merged (green dotted arrow) with the previous context without any loss of precision, as formalised in Theorem 1.

Our algorithm now propagates backwards the dependency sets, interpolants and witness paths to the previous branch on `d`, resulting in the set  $\{d, x\}$ , interpolant  $flag = 1$ , and witness path  $d \wedge y = 4 \wedge flag \wedge z = y + x$ . Note that this time, it considered the control-dependence of `y` on `d` as both paths from the `d` branch were feasible, thus adding `d` to the dependency set. It then backtracks to the first split point on `c` and explores the other branch `!c`. Upon reaching the branch on `d` again, it tries to merge with the previous context of the `d` branch by checking if the current path formula  $\neg c \wedge flag = 0 \wedge x = 2$  implies the interpolant  $flag = 1$ . It does not, so the merging cannot be performed and so it proceeds to explore the rest of the tree under the node, resulting in the final SE tree as shown in Fig. 3.5(b).

There are a few important things to note in the final tree. The branch on `d` is duplicated due to the split at the previous branch on `c`. However under the context `!c`, the dependency set at the branch point on `flag` is only  $\{x\}$  as opposed to  $\{y, x\}$  under the context `c`. This is the advantage of path-sensitivity – we have obtained a more precise dependency information at a different context of the same program point by considering the contexts separately, although at the price of duplication of the `d` branch. However we will see soon that because

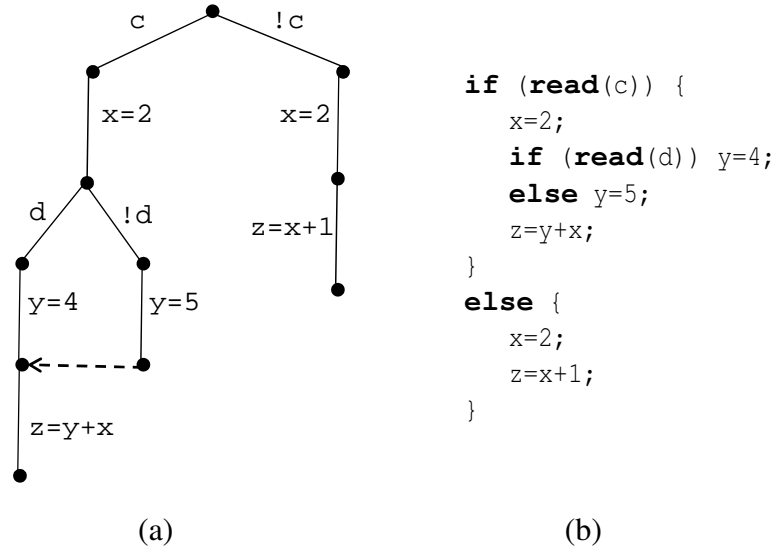


Figure 3.6: The PSS-CFG and corresponding transformed program for Fig. 3.5

of the more precise information, the duplication can be controlled by slicing.

## Phase Two: Transformation of the SE tree

We have defined a set of rules that will process the SE tree annotated with dependency information to transform it into the final PSS-CFG. We give an informal description of each rule here, as they are formalised in Section 3.10.2.

- **Rule 1** states if the LHS of an assignment statement does not occur in the dependency set after it, the statement can be removed. This is the traditional slicing rule for assignment statements.
- **Rule 2** states that if a branch point has only one feasible path arising from it, the branch point can be removed. The reasoning is that if a branch point has only one feasible path from it, then in that particular context the branch condition can be deterministically evaluated to true (or false). Thus it can simply be replaced with the “then” (or “else”) body.
- **Rule 3** (called “Tree Slicing”), which is more powerful in reducing the PSS-CFG’s size, states that an entire branch is irrelevant to the target and can be removed if both the “then” and “else” bodies contain no statement that is included in the slice. This rule is more complicated than it seems at first because working with trees, a problem arises when we remove a branch point: conceptually there could be *two* sub-trees whose parent, the

branch point, is about to be removed. The two sub-trees could be arbitrarily different because of the different contexts leading into them. Which one should be linked to the branch point's parent? The rule guarantees that regardless of which sub-tree is picked, the transformation is still sound, *provided* that our algorithm declared the sub-trees to be merged. This important non-trivial result is formalised in Section 3.10.2, and is one of the many fundamental differences between our transformation method and *static slicing* methods, that slice on the program, not the tree.

Note that in general, the rules are not limited to the above three, and one can indeed formulate more sophisticated rules. But for our benchmarks these rules were sufficient to provide benefit. These rules are applied on the SE tree until none of them can be applied any more (fixpoint), and the resultant graph is the PSS-CFG.

In our example, applying Rule 1 on the SE tree removes the statements `flag=1` and `flag=0`. Applying Rule 2 then removes the two branches on `flag` that have an infeasible path. More interesting is the application of Rule 3. It cannot be applied on the `d` branch under the context `c` because in that context, `y=4` and `y=5` will be included in the slice (recall that the dependency set after the branch is  $\{y,x\}$ ). However, it can indeed be applied on the `d` branch under the context `!c` because neither `y=4` nor `y=5` is included in the slice (recall that the dependency set after the `d` branch in this context is only  $\{x\}$ ), and our algorithm had merged the symbolic state after its “then” and “else” body.

Thus, Rule 3 removes the `d` branch under the context `!c` to get the final PSS-CFG in Fig. 3.6(a). This reduction of the graph due to slicing counteracts the blow-up due to path-sensitivity, and is critical to maintaining the size of the PSS-CFG. Finally, note that Rule 3 cannot be applied on the top-level `c` branch because the two subtrees after its “then” and “else” body have not been merged. This means the split due to the `c` branch is causing some differences in the two subtrees related to the target, and hence removing the branch could make the PSS-CFG incorrect. Indeed, the `c` branch assigns different values to `flag` which ultimately causes different values to be assigned to the target `z`. Thus the branch must be kept to preserve the original program's semantics.

Finally, as a third “step” of our algorithm, we produce an equivalent C program from the PSS-CFG. The transformation process is quite straightforward so we do not detail it here. It is done primarily so that external off-the-shelf applications can be executed on the PSS-CFG. The transformed program for our example is shown in Fig. 3.6(b).

At the outset, one can notice that the transformed program has only 3 paths compared to 8 paths in the original program. Moreover, information that cannot be captured from the original program can be captured by the transformed program. For instance, a concolic tester on the original program will always generate a value for `d` regardless of the value generated for `c`. However in the transformed program, if the value of `c` was generated to be 0, the tester would not generate the value of `d` because it will not affect the target `z`.

It can also be seen that the variable `flag`, which was mainly used for control flow between different parts of the code, is not even present in the transformed program. This information cannot be captured by static slicers like [29, 65], which cannot statically remove the assignments to `flag` or the branch on `flag` from the program without becoming unsound.

**Remark.** One might wonder if our complete algorithm to produce the PSS-CFG is equivalent to simply expanding the paths of the original program producing a semantically equivalent program, deleting the infeasible paths, and applying standard slicing w.r.t. the target. Even though conceptually it may be similar, there are many practical differences with our method. Without our algorithm’s merging, one would run into exponential blowup of paths during symbolic execution, before even producing the semantically equivalent program.

Even if a merging mechanism is used to contain the blowup, without the guarantee of *lossless* merging provided by our algorithm, one could obtain imprecise dependency information thereby keeping irrelevant statements in the new program. However, our algorithm provides the right balance between precision and performance of such a target-based transformation. Thus, the process of constructing the SE tree and the process of dependency computation are closely intertwined and cannot be separated and outsourced to an external slicer.

## 3.9 Background

**Tree transformation rules.** In addition to the formalisms for dependency computation in Section 3.2, we formalise the representation of the SE tree to be conveniently processed by the transformation rules. The SE tree produced by our algorithm, together with the dependency information of each symbolic state, is represented using the set  $\mathcal{S}$  of *facts* of the following types:

- $\text{edge}(v \xrightarrow{\text{op}} v')$ , denoting a feasible edge from  $v$  to  $v'$
- $\text{inf\_edge}(v \xrightarrow{\text{op}} v')$ , denoting an infeasible edge from  $v$  to  $v'$
- $\text{merged}(v, v')$ , denoting that  $v$  has been merged with  $v'$  (will be formalised later)
- $\text{in\_slice}(v \xrightarrow{\text{op}} v')$ , denoting that the transition from  $v$  to  $v'$  is included in the slice due to Eqs. 3.1,3.4.

Note that we do not explicitly store the dependency information at each state, but rather just the fact whether a transition from the state is included in the slice or not (denoted by the  $\text{in\_slice}$  fact). In Section 3.10.2, the transformation of the SE tree into the final PSS-CFG will be modelled using certain rules that act upon these facts.

## 3.10 Algorithm

We describe our algorithm in two phases: in phase one (Section 3.10.1), we explore symbolic paths in the program to generate the symbolic execution (SE) tree annotated with dependencies. In phase two (Section 3.10.2), we transform this tree by removing edges and sub-trees, to finally produce the PSS-CFG.

At a high level, our algorithm performs forward symbolic execution, as usual in a depth-first manner interleaved with backward dependency computation. Symbolic execution avoids the exploration of infeasible paths, thus increasing the precision of the computed dependencies. However, it allows multiple copies of the same program point to exist as different *symbolic states*, along different symbolic paths. Thus an important challenge again to overcome is to avoid this inherent exponential blowup of symbolic execution. Our solution is to merge different symbolic states provided certain conditions are met. These merging

conditions, dictated by interpolants and witness paths, (defined in Definition 3), guarantee that the merge does not incur any loss of slicing information.

We now provide the important proof of Theorem 1 (restated below), given that the analysis information at each symbolic state  $\nu$  is a set of dependency variables  $\sigma_\nu$ .

**Theorem 2.** *Given states  $\nu \equiv \langle \ell, s, \Pi \rangle$  and  $\nu' \equiv \langle \ell, s', \Pi' \rangle$ , let  $\sigma_{\nu'}$  be the dependencies and witness formulas associated with  $\nu'$ . If  $\nu$  can be merged with  $\nu'$  then by exploring  $\nu$  there cannot be produced a set of dependencies  $\sigma_\nu$  such that  $\sigma_\nu \neq \sigma_{\nu'}$ .*

PROOF. *Assume that although  $\nu$  can be merged with  $\nu'$  (i.e., both conditions of Eqn. 2.2 are satisfied), it is instead symbolically explored and a dependency set  $\sigma_\nu$  is obtained.*

**Proof that  $\sigma_{\nu'} \subseteq \sigma_\nu$ :** *Since  $\nu$  can be merged with  $\nu'$ , by condition (b) of Eqn. 2.2,  $\forall \langle x, \cdot \rangle \in \sigma_{\nu'}$ , there is a witness path, say  $\pi_x$ , with formula  $\omega_x$  such that  $\llbracket \nu \rrbracket \wedge \omega_x$  is satisfiable. That is,  $\pi_x$  is feasible from  $\nu$ . By the definition of a witness path (Definition 2),  $\exists v_1 \in \mathcal{V}$  s.t  $v_1$  is control- or data-dependent on  $x$  along the path  $\pi_x$ , which is feasible from  $\nu$ . Therefore  $x$  must be in  $\sigma_\nu$ .*

**Proof that  $\sigma_\nu \subseteq \sigma_{\nu'}$ :** *(by contradiction) Assume  $\exists x \in \sigma_\nu$  s.t  $x \notin \sigma_{\nu'}$ . Then, the witness path for  $x$ ,  $\pi_x$  with formula  $\omega_x$  must be such that  $\llbracket \nu \rrbracket \wedge \omega_x$  is satisfiable but  $\llbracket \nu' \rrbracket \wedge \omega_x$  is unsatisfiable (otherwise from the definition of a witness path,  $x$  would have been included in  $\sigma_{\nu'}$ ). That is,  $\pi_x$  is feasible from  $\nu$  but infeasible from  $\nu'$ . From Eqn. 2.2 condition (a) and Lemma 1, this is impossible.  $\square$*

### 3.10.1 Generating the SE tree structure with dependencies

We now describe our main algorithm GENPSSCFG in Fig. 3.7. The purpose of GENPSSCFG is to generate a finite complete symbolic execution tree annotated with dependency information at each symbolic state. As mentioned in

```

GENPSSCFG ( $v \equiv \langle \ell, s, \Pi \rangle$ )
1: if  $\exists v' \equiv \langle \ell', s', \Pi' \rangle$  s.t.  $v$  and  $v'$  satisfy Eqn. 2.2
2:   then MERGE ( $v, v'$ )
3: else if  $v$  is at a branch point then
4:   SPLIT( $v$ )
5: else
6:   SYMEXEC ( $v$ )

MERGE ( $v, v'$ )
1:  $\bar{\Psi}_v = \bar{\Psi}_{v'}$ 
2:  $\sigma_v = \sigma_{v'}$ 
3:  $\mathcal{S} = \mathcal{S} \cup \text{merged}(v, v')$ 

SPLIT ( $v \equiv \langle \ell, s, \Pi \rangle$ )
1:  $\bar{\Psi}_v = \text{true}$ 
2: foreach transition  $\ell \xrightarrow{\text{assume}(c)} \ell'$  do
3:   if ( $v$  is a loop header) then
4:      $v' \triangleq \langle \ell', \cdot, \text{invariant}(v) \wedge \llbracket c \rrbracket_s \rangle$ 
5:   else
6:      $v' \triangleq \langle \ell', s, \Pi \wedge \llbracket c \rrbracket_s \rangle$ 
7:   if  $v'$  is infeasible state then
8:      $\mathcal{S} = \mathcal{S} \cup \text{inf\_edge}(v \xrightarrow{\text{assume}(c)} v')$ 
9:      $\bar{\Psi}_{v'} = \text{false}, \sigma_{v'} = \emptyset$ 
10:  else
11:     $\mathcal{S} = \mathcal{S} \cup \text{edge}(v \xrightarrow{\text{assume}(c)} v')$ 
12:    GENPSSCFG ( $v'$ )
13:     $\bar{\Psi}_v = \bar{\Psi}_v \wedge \widehat{wlp}(\bar{\Psi}_{v'}, \text{assume}(c))$ 
14:     $\sigma_v = \sigma_v \sqcup \widehat{pre}(\sigma_{v'}, \text{assume}(c), s)$ 
15:    if  $\delta \equiv v \xrightarrow{\text{assume}(c)} v'$  satisfies Eqn. 3.4 then
16:       $\mathcal{S} = \mathcal{S} \cup \text{in\_slice}(v \xrightarrow{\text{assume}(c)} v')$ 

SYMEXEC ( $v \equiv \langle \ell, s, \Pi \rangle$ )
1: if  $\nexists$  transition relation  $\ell \xrightarrow{x:=e} \ell'$  then
2:    $\bar{\Psi}_v = \text{true}, \sigma_v = \mathcal{V}$ 
3: else
4:    $v' \triangleq \langle \ell', s[x \mapsto \llbracket e \rrbracket_s], \Pi \rangle$ 
5:    $\mathcal{S} = \mathcal{S} \cup \text{edge}(v \xrightarrow{x:=e} v')$ 
6:   if  $v'$  is not a loop header
7:     GENPSSCFG ( $v'$ )
8:    $\bar{\Psi}_v = \widehat{wlp}(\bar{\Psi}_{v'}, x:=e)$ 
9:    $\sigma_v = \widehat{pre}(\sigma_{v'}, x:=e)$ 
10:  if  $v \xrightarrow{x:=e} v'$  satisfies Eqn. 3.1 then
11:     $\mathcal{S} = \mathcal{S} \cup \text{in\_slice}(v \xrightarrow{x:=e} v')$ 

```

Figure 3.7: Symbolic execution interleaved with dependency computation to produce the SE tree

Section 3.9, the tree is represented using a set of facts that are added to the set  $\mathcal{S}$ , which is assumed to be a global variable to the algorithm.

GENPSSCFG requires the program to have been translated to a transition system  $\langle \Sigma, I, \longrightarrow, O \rangle$  in SSA form [30], and accepts a symbolic state as argument. It is initiated with the state  $v \equiv \langle \ell_{\text{start}}, \varepsilon, \text{true} \rangle$ . GENPSSCFG implements a mutually recursive algorithm with a few other procedures.

First, the most important decision of whether to merge a symbolic state with another is taken by GENPSSCFG at line 1. It attempts to find another symbolic state  $v'$  such that  $v$  and  $v'$  satisfy the two merging conditions in Equation 2.2. If yes, it merges  $v$  with  $v'$  by calling the procedure MERGE at line 2. If such a  $v'$

does not exist, GENPSSCFG decides whether to split the symbolic execution of  $\nu$  or not by checking if  $\nu$  corresponds to a branching point in the program (line 3). If yes it calls the procedure SPLIT at line 4 which, as we will see, forks the symbolic execution of different branches from  $\nu$ . If both the above cases do not match, GENPSSCFG simply continues the symbolic execution by calling the procedure SYMEEXEC with  $\nu$ . GENPSSCFG is in essence the high level backbone of our method.

The procedure MERGE, given a current symbolic state  $\nu$  and an already explored state  $\nu'$ , merges the former with the latter by setting the interpolant and dependency set of  $\nu$  to those of  $\nu'$ . Recall that Theorem 2 guaranteed such a merge to have no loss of precision. That is, had  $\nu$  been explored instead of being merged with  $\nu'$ , the resulting dependency set at  $\nu$  would be exactly  $\sigma_\nu$ . Finally the procedure adds the fact  $\text{merged}(\nu, \nu')$  to  $\mathcal{S}$  to record the merge between the two states.

The procedure SPLIT is used to fork the symbolic execution of a state from which multiple transitions are possible (typically a branch point). Given a symbolic state  $\nu$  with program point  $\ell$  and path condition  $\Pi$ , it first initialises its interpolant  $\bar{\Psi}_\nu$  to true at line 1. At line 2 it iterates its main body over each transition possible from  $\nu$ . Now there is an issue: if the current state is a loop header (line 4), then symbolically executing the loop could result in an *unbounded* tree, which we want to avoid. Therefore, we need to execute the loop with a *loop invariant* to make the tree finite.

Our method to compute a loop invariant is simple but effective: from the loop header's symbolic state  $\nu$ , we only keep the constraints that are unchanged through the loop, and delete the rest. For instance, if  $x > 5$  holds at the loop header and  $x$  is only incremented in the loop, then  $x > 5$  is unchanged through the loop. This widened state at  $\nu$  ultimately forms a loop invariant. This technique provides a balance between getting the strongest invariant – which is needed to maximise path-sensitivity – and efficiency. We found experimentally that this technique preserves most of the important information through the loop. Nevertheless, we remind the reader that no matter what the invariant is, it does not affect the guarantee of lossless-ness of dependency information during our



merging, and the correctness of our transformation as stated by Theorem 3.

We assume a function invariant that given a symbolic state  $\nu$ , returns a FOL formula representing the loop invariant. With this invariant, the next state is constructed by augmenting it with  $\llbracket c \rrbracket_{\mathcal{S}}$  where  $c$  is the branching condition of the assume statement (line 4). If not,  $\nu'$  is constructed (line 6) by augmenting the path condition  $\Pi$  with  $\llbracket c \rrbracket_{\mathcal{S}}$ . At line 7 an important check is performed: if  $\nu'$  is an infeasible state (i.e., the augmented path condition is unsatisfiable), it means symbolic execution has encountered an infeasible path. Therefore it adds to  $\mathcal{S}$  the fact that the transition from  $\nu$  to  $\nu'$  is infeasible (line 8), and sets the interpolant and dependency set of  $\nu'$  to false and  $\emptyset$  respectively (line 9) to signify that the state is unreachable. Otherwise it adds a normal edge to  $\mathcal{S}$  at line 11 and (mutually) recursively calls GENPSSCFG with  $\nu'$ .

In either case,  $\nu'$  would have been annotated with an interpolant  $\bar{\Psi}_{\nu'}$  and dependency set  $\sigma_{\nu'}$ . Now it computes the same information for  $\nu$  at lines 13-14. The interpolant  $\bar{\Psi}_{\nu}$  is supposed to generalise the SE tree below  $\nu$  while preserving its infeasible paths. For this, the procedure  $\widehat{wlp} : FOL \times Ops \rightarrow FOL$  is called that ideally computes the *weakest liberal precondition* [36], the weakest formula on the initial state ensuring the execution of `assume(c)` results in the state  $\bar{\Psi}_{\nu'}$ . In practice we approximate  $wlp$  by making a linear number of calls to a theorem prover following techniques described in [66], usually resulting in a formula stronger than the weakest liberal precondition. The dependency set  $\sigma_{\nu}$  is computed by applying the pre-operation  $\widehat{pre}$  on  $\sigma_{\nu'}$  and joining with any existing set (across different iterations of the main loop).

Finally, in lines 15-16 of SPLIT, it checks if any transition from  $\delta$  to its nearest postdominator is included in the slice (Eqn. 3.4). If yes, it adds an `in_slice` fact to  $\mathcal{S}$  with the transition from  $\nu$  to  $\nu'$ .

The final procedure SYMEXEC is called by GENPSSCFG when the current symbolic state  $\nu$  corresponding to program point  $\ell$  cannot split (typically an assignment statement). Initially, at line 1, it checks if there exists a program transition from  $\ell$  to any other  $\ell'$ . If not, symbolic execution has reached the end of a (feasible) path whose final state is  $\nu$ . In other words, it has reached a terminal node. Hence it sets the interpolant  $\bar{\Psi}_{\nu}$  to true and its dependency set

$\sigma_{\nu}$  to  $\mathcal{V}$  (recall that the target variables are specified at  $\ell_{\text{end}}$ ) at line 2.

If there exists a transition from  $\ell$  to say  $\ell'$  with the assignment  $x:=e$ , it constructs the next symbolic state (line 4)  $\nu'$  by setting in the store  $s$  the value of  $x$  to  $\llbracket e \rrbracket_s$  and adds to  $\mathcal{S}$  the appropriate edge fact (line 5). Then, if  $\nu'$  is not a loop header, it recursively calls GENPSSCFG with  $\nu'$  (line 7). If  $\nu'$  is a loop header, then there is no need to explore it again since it would have already been explored with the loop invariant (at SPLIT line 4). Our algorithm thus makes the symbolic execution finite. In SYMEXEC line 8 and 9, it sets the interpolant (and dependency set) of  $\nu$  by calling  $\widehat{wlp}$  (and  $\widehat{pre}$ ) on the interpolant (and dependency set) of  $\nu'$ . Finally, at lines 10-11, if  $x$  contains a variable in  $\sigma_{\nu'}$  (Eqn. 3.1) it adds to  $\mathcal{S}$  the fact that the transition from  $\nu$  to  $\nu'$  is included in the slice.

To perform the fixpoint computation at the highest level, we keep making calls to GENPSSCFG until there is no change in  $\mathcal{S}$ . This is the simplest way to describe the fixpoint computation but in practice we can optimise it by calling GENPSSCFG with the symbolic state of the loop header in which the change was detected.

### 3.10.2 Transformation of the annotated SE tree

The algorithm described so far produces a symbolic execution tree represented as a set of facts  $\mathcal{S}$ . Now we present certain rules in Fig. 3.8 that act upon  $\mathcal{S}$  to modify it, in essence modelling the transformation of the SE tree into the final PSS-CFG. The rules are presented in a declarative fashion and can be implemented conveniently in a rule-based programming language (e.g., Constraint Handling Rules).

**STRAIGHT LINE SLICING:** This rule states that if there is a transition (or edge) from state  $\nu_0$  to  $\nu_1$  and an *assignment* transition from  $\nu_1$  to  $\nu_2$  such that the latter is not included in the slice, then both transitions can be removed and replaced with one linking  $\nu_0$  directly to  $\nu_2$ . This is the typical rule for slicing assignment statements using dependencies.

**INFEASIBLE PATH REMOVAL:** This rule states that if there is a transition from

RULE 1 (STRAIGHT LINE SLICING)

$$\frac{E_1 \equiv \text{edge}(v_0 \xrightarrow{\text{op}} v_1) \in \mathcal{S} \quad E_2 \equiv \text{edge}(v_1 \xrightarrow{x:=e} v_2) \in \mathcal{S} \quad \text{in\_slice}(v_1 \xrightarrow{x:=e} v_2) \notin \mathcal{S}}{\mathcal{S} = \mathcal{S} \setminus \{E_1, E_2\} \cup \{\text{edge}(v_0 \xrightarrow{\text{op}} v_2)\}}$$

RULE 2 (INFEASIBLE PATH REMOVAL)

$$\frac{E_1 \equiv \text{edge}(v_0 \xrightarrow{\text{op}} v_1) \in \mathcal{S} \quad E_2 \equiv \text{edge}(v_1 \xrightarrow{\text{assume}(c_1)} v_2) \in \mathcal{S} \quad E_3 \equiv \text{inf\_edge}(v_1 \xrightarrow{\text{assume}(c_2)} v_3) \in \mathcal{S}}{\mathcal{S} = \mathcal{S} \setminus \{E_1, E_2, E_3\} \cup \{\text{edge}(v_0 \xrightarrow{\text{op}} v_2)\}}$$

RULE 3 (TREE SLICING)

$$\frac{\text{edge}(v_0 \xrightarrow{\text{op}} v_1) \in \mathcal{S} \quad \text{edge}(v_1 \xrightarrow{\text{assume}(c_1)} v_2) \in \mathcal{S} \quad \text{edge}(v_1 \xrightarrow{\text{assume}(c_2)} v_3) \in \mathcal{S} \quad v_2 \neq v_3 \quad \text{in\_slice}(v_1 \xrightarrow{\text{assume}(c_1)} v_2) \notin \mathcal{S} \quad \text{in\_slice}(v_1 \xrightarrow{\text{assume}(c_2)} v_3) \notin \mathcal{S} \quad \langle v_k, v'_k \rangle \equiv \text{MergePoint}(v_1) \quad \text{merged}(v_k, v'_k) \in \mathcal{S}}{\mathcal{S} = \mathcal{S} \setminus \{\text{edge}(v' \xrightarrow{\text{op}} v'') \mid v' \xrightarrow{\text{op}} v'' \in \text{INFL}(v_1 \xrightarrow{\text{assume}(c_1)} v_2) \vee v' \xrightarrow{\text{op}} v'' \in \text{INFL}(v_1 \xrightarrow{\text{assume}(c_2)} v_3)\} \cup \{\text{edge}(v_0 \xrightarrow{\text{op}} v_k)\}}$$

Figure 3.8: Transformation rules to produce the final PSS-CFG

state  $v_0$  to  $v_1$ , and  $v_1$  is a *branch point* such that there is branching edge (*edge*) from  $v_1$  to  $v_2$  and an infeasible branching edge (*inf\_edge*) from  $v_1$  to another  $v_3$ , then all three edges can be removed and  $v_0$  can be directly linked to the feasible state  $v_2$ .

**TREE SLICING:** This rule is more complicated and the most powerful in terms of reducing the symbolic state space of the PSS-CFG. It states that if there is a transition from  $v_0$  to  $v_1$ , and  $v_1$  is a branching point with branching transitions to  $v_2$  (with condition  $\text{assume}(c_1)$ ) and  $v_3$  (with condition  $\text{assume}(c_2)$ ) such that neither transition is included in the slice, then we can remove all transitions  $v' \xrightarrow{\text{op}} v''$  that occur either in the dynamic range of influence (given by *INFL*) of  $v_1 \rightarrow v_2$  or  $v_1 \rightarrow v_3$ . In other words, we can remove all transitions that occur in the “then” or “else” body of the branch at  $v_1$ . But there is a problem:

since we are working on a symbolic tree, removing the branch point  $v_1$  would conceptually leave two different subtrees “hanging” without a parent. The question arises as to which subtree should we link to the node  $v_0$ . TREE SLICING guarantees that if the symbolic states at the end of the branch  $\langle v_k, v'_k \rangle$  (as returned by  $\text{MergePoint}(v_1)$ ) are merged by our algorithm (i.e.,  $\text{merged}(v_k, v'_k)$  exists), the differences in the trees do not affect the target variables. Hence it simply adds a transition directly linking  $v_0$  to one of the symbolic states  $v_k$ .

We explain the reasoning behind the above rules by defining our correctness statement for the transformation of the SE tree into the PSS-CFG and providing a proof outline for it. For an example of application of the rules, refer to Section 3.8. First let two CFGs be defined *equivalent* w.r.t. target variables  $\mathcal{V}$  if for any input, the programs corresponding to both CFGs produce the same values for all variables in  $\mathcal{V}$ .

**Theorem 3.** (*Correctness of transformation*) *An application of RULE 1, RULE 2 or RULE 3 to a CFG  $G$  produces a transformed CFG  $G'$  such that  $G'$  is equivalent to  $G$  w.r.t. target variables  $\mathcal{V}$ .*

We now provide a proof outline for Theorem 3. The correctness of STRAIGHT LINE SLICING follows directly from the correctness of slicing assignment statements using dependency information, formalised in Eqn. 3.1. As for INFEASIBLE PATH REMOVAL, for any input that executes a path in  $G$  leading to the state  $v_1$ , the condition  $c_1$  will evaluate to true and the condition  $c_2$  will evaluate to false. Moreover, an `assume` statement does not modify any variable in the program state. Thus, both checks  $\text{assume}(c_1)$  and  $\text{assume}(c_2)$  are useless because we deterministically know their outcomes, and hence can be replaced with a transition linking  $v_0$  to the next feasible state  $v_2$  to produce  $G'$ .

The correctness proof of TREE SLICING is as follows. Assume that some input executes a path in  $G$  starting from  $v_{\text{start}}$  to  $v_0$  and then reaches  $v_1$ . W.l.o.g, assume that the condition  $c_1$  holds at  $v_1$ , therefore it chooses to follow  $v_2$ , reaches the merged point  $v_k$  and continues to eventually reach the terminal state  $v_{\text{end}}$ . Let us call this executed path  $\pi_G$ . In  $G'$ , obtained by applying TREE SLICING on  $G$ , thereby removing the entire branch at  $v_1$ , the same input would

follow a path, say  $\pi_{G'}$ , such that  $\pi_{G'}$  is the exact same path as  $\pi_G$  starting from  $\nu_{\text{start}}$  till  $\nu_0$ , thus having the same symbolic state at  $\nu_0$ . At this point,  $\pi_{G'}$  differs from  $\pi_G$  by implicitly “skipping” the execution of the branch at  $\nu_1$  and instead directly reaches  $\nu_k$ .

Since  $\nu_k$  and  $\nu'_k$  were merged, the dependency sets at both points are the same. Now, since the transition  $\nu_1 \xrightarrow{\text{assume}(c_1)} \nu_2$  in  $G$  was not included in the slice, it means that no statement “skipped” by  $\pi_{G'}$  affected the dependency information at  $\nu_k$ . This implies that the symbolic state of the path  $\pi_{G'}$  at  $\nu_k$  is the same as the symbolic state of the path  $\pi_G$  at  $\nu_k$  *as far as the dependency variables at  $\nu_k$  are concerned*. To be precise, the values of the dependency variables at  $\nu_k$  are the same in both  $\pi_G$  and  $\pi_{G'}$ . Since these are the only variables affecting the target variables  $\mathcal{V}$  at  $\nu_{\text{end}}$ , it is sufficient to preserve their values to ensure that  $\pi_{G'}$  will produce the same values for  $\mathcal{V}$  as  $\pi_G$ . Of course  $\pi_{G'}$  may produce different values than  $\pi_G$  for variables *not* in  $\mathcal{V}$ , but we are not interested in those variables.  $\square$

The three rules are applied until fixpoint is reached (i.e., none of them can be applied anymore). Termination of rule applications is guaranteed from the initial finiteness of the set  $\mathcal{S}$  and the fact that all three rules remove more edges from  $\mathcal{S}$  than they add. Soundness of individual rule applications is guaranteed from Theorem 3. Transitivity of the rules is also guaranteed by Theorem 3 since each new CFG is equivalent to the previous CFG. Once fixpoint is reached, the final PSS-CFG structure can be extracted from  $\mathcal{S}$ .

Thus, Theorem 3 guarantees that the PSS-CFG is equivalent to the original program w.r.t. the target variables  $\mathcal{V}$ . Therefore, any analysis of the original program concerned only with  $\mathcal{V}$  can be applied on the PSS-CFG instead to take advantage of its benefits. In the next section, we will see two such applications: program testing and verification.

Benchmark	Lines of code			Blow up	PSS Time	#Rule Triggers		
	Orig	St.slice	PSS			Rul1	Rul2	Rul3
cdaudio	1817	1599	4452	2.78	24s	2685	1101	169
diskperf	937	706	2967	4.20	18s	1594	1132	73
floppy	1005	766	2086	2.72	7s	1062	651	99
floppy2	1513	1250	3507	2.81	16s	1514	819	120
kbfiltr	549	275	170	0.62	1s	111	46	7
kbfiltr2	782	492	410	0.83	1s	249	69	23
tcas	286	227	311	1.37	2s	138	204	47

Table 3.2: Statistics about the PSS-CFG

### 3.11 Experimental Evaluation

We evaluate the PSS-CFG using applications of program testing and verification to show considerable increase in their performance. We implemented the algorithm described in Section 3.10 on the TRACER [64] framework for symbolic execution. We used as benchmarks device drivers from the ntdrivers-simplified category of SV-COMP 2012 [11], and a traffic collision avoidance program called tcas, and chose the target variables from the safety properties of the programs. All programs had multiple safety properties on several variables, all of which were used in the slicing criterion. For practically applying external tools on the PSS-CFG structure, we used its equivalent decompiled program. Since both the original and decompiled programs are in C, we can easily apply external tools on them and measure how their benefit from our transformation.

For all our experiments we compare the PSS-CFG<sup>10</sup> with a static slice of the benchmark program on the target variables. Comparing with a static slice is more challenging as some statements would have already been sliced away from the original program. We obtained the static slice through the well-known state-of-the-art slicer Frama-C [29, 1]. Frama-C is a path-sensitive static slicer that can detect infeasible paths through techniques such as constant propagation, constant folding and abstract interpretation. Also, before the target variables are provided and our algorithm is initiated, we process the program and store an intermediate representation (IR). This processing involves computing information about infeasible paths in the program and is completely independent of the

<sup>10</sup>We use the terms “decompiled program” and PSS-CFG interchangeably.

target variables. Then, when the target variables are provided, our algorithm is invoked and it uses information from this IR. All experiments were run on an Intel 3.2 Ghz system with 2GB memory.

Now, we provide statistics about the PSS-CFG and its construction in Table 3.2. The Lines of code column shows the number of non-commented lines of code in the original (Orig) program, its static slice (St.slice) and its decompiled program (PSS) respectively. In the column Blowup we show the ratio of the LOC of PSS-CFG compared to the static slice. The blowup is a result of the balance between the splits introduced by path-sensitivity, and the merges and slicing from our algorithm. It is clear that the blowup is manageable, sometimes even smaller than the program, being on average around 2. In the column PSS Time we show the time taken in seconds for our algorithm to produce the PSS-CFG given the target variables, which is modest. In the final column #Rule Triggers we show the number of times each transformation rule was triggered during PSS-CFG construction. Although RULE 3 is shown to be triggered fewer number of times than RULE 1 or RULE 2, it is the most powerful rule in reducing the search space of the PSS-CFG. In *tcas* we see RULE 2 triggering more frequently than RULE1 due to its large number of infeasible paths.

Note that the PSS-CFG construction is only performed once for a given set of target variables. The resulting program can however be subjected to an innumerable number of properties to be verified or tested. For example, using the same PSS-CFG, one can verify different bounds on a target variable depending on different preconditions to the program.

### 3.11.1 Testing

We consider software testing an important application for the PSS-CFG to be used. For this, we consider the typical DART [49] methodology that performs *concolic testing*, i.e., executing the program with both concrete and symbolic inputs and symbolically negating branches to explore new paths. We chose the publicly available concolic tester CREST, an implementation of DART for C programs. Since the statically sliced and decompiled programs are in C, the experiment was simply to run the concolic testing process on both programs

Benchmark	Testing Time		Speed up	#Solver calls	
	St.slice	PSS		St.slice	PSS
cdaudio	1m30s	43s	2.1	16k	7k
diskperf	900m	34m	26.5	26mil	1mil
floppy	9m6s	24s	22.8	260k	4k
floppy2	525m	429m	1.2	613k	479k
kbfiltr	2s	1s	2	63	52
kbfiltr2	22s	6s	3.7	7k	2k
tcas	4s	1s	4	1.5k	188
<b>Total</b>	<b>23h56m</b>	<b>7h44m</b>	<b>3.1</b>	<b>26.9mil</b>	<b>1.5mil</b>

Table 3.3: Experiments on the PSS-CFG for concolic testing

and measure the time taken to complete, i.e., time taken to test all feasible paths in the program.

In Table 3.3, we show the measures of the experiment. The second and third columns (St.slice and PSS) show the time taken to complete the concolic testing process on the statically sliced and decompiled programs respectively. The third column shows the Speedup obtained by using the PSS-CFG, i.e., the ratio of the columns St.slice and PSS. It is immediately apparent that the PSS-CFG provides speedup in all benchmarks. In programs diskperf and floppy the speedup is exceptionally high around 22-26, reducing the concolic testing time from, for instance, 900 minutes (15 hours) to just 34 minutes. On the other hand, in floppy2 the speedup of 1.2 is not that high, but still the absolute benefit in time can be seen – around 96 minutes or 1.5 hours. Ultimately, the total time taken for concolic testing to run on all our statically sliced programs was almost 24 hours, whereas it took less than 8 hours to run on the decompiled programs, providing a net benefit in time of a magnitude of 3.1. Although it is understood that in practice concolic testing may not terminate by exploring all paths, we gave a huge timeout (24 hours) for the process to terminate simply to see how much benefit the PSS-CFG can provide in timing. From the table, it is clear that the PSS-CFG can make the difference between termination and timing-out of the concolic testing process.

In addition to time, we also measured the number of calls made by CREST to its underlying solver. This measure, shown in the column #Solver calls, gives an idea of how the PSS-CFG would still benefit the concolic tester even if a



Benchmark	IMPACT			ARMC			CPA-CHECKER		
	Verification Time		Speed	Verification Time		Speed	Verification Time		Speed
	St.Slice	PSS	up	St.Slice	PSS	up	St.Slice	PSS	up
cdaudio	95s	14s	6.8	T/O	21s	N/A	26s	14s	1.86
diskperf	146s	18s	8.1	T/O	6s	N/A	7s	6s	1.17
floppy	34s	8s	4.3	259s	6s	43.17	6s	5s	1.20
floppy2	39s	13s	3.0	T/O	17s	N/A	10s	8s	1.25
kbfiltr	4s	1s	4.0	3s	1s	3.00	3s	2s	1.50
kbfiltr2	8s	2s	4.0	13s	2s	6.50	4s	2s	2.00
tcas	3s	1s	3.0	3s	1s	3.00	2s	1s	2.00
<b>Total</b>	<b>329s</b>	<b>57s</b>	<b>5.8</b>	<b>T/O</b>	<b>54s</b>	<b>N/A</b>	<b>58s</b>	<b>38s</b>	<b>1.53</b>

(a)

(b)

(c)

Table 3.4: Experiments on the PSS-CFG for verification

different, faster solver was used. Again we see several magnitudes of less solver calls for all benchmarks when CREST was run on the PSS-CFG. This is because of the considerably less number of paths the tester has to execute, in turn leading to less constraint solving. The maximum benefit is in `diskperf` where 26 million calls were made for the statically sliced program, compared to only 1 million for the decompiled program. This is in-line with the speedup in time for `diskperf`, around 26. This indicates that even if a faster solver is used, the relative speedup in time for this benchmark would still be around 26, although the absolute timings may be faster. Ultimately, this table shows that concolic testing would definitely benefit by using the PSS-CFG instead of the statically sliced program.

### 3.11.2 Verification

Another important application for the PSS-CFG is program verification. In Table 3.4 we compare the verification times of the benchmarks across three different state-of-the-art verifiers: IMPACT [80], ARMC [87] and CPA-CHECKER [14], each representing different approaches to verification—interpolant-based, CEGAR-based and SMT-based. Since IMPACT is not publicly available, we use CPA-CHECKER’s implementation of the IMPACT algorithm. In each table, the second and third columns show the verification time (in seconds) of the statically sliced program (St.Slice) and the PSS-CFG (PSS), respectively. In the

third column Speedup, we show the ratio of St.slice to PSS.

For all three verifiers, it can be clearly seen that the PSS-CFG is verified in a much faster time than the static slice. For IMPACT, verifying all statically sliced programs in our suite took 329 seconds whereas verifying the respective PSS-CFGs took only 57 seconds. Thus, the speedup across all programs on aggregate is 5.8. As for ARMC, it was unable to terminate its verification of the statically sliced programs for `cdaudio`, `diskperf` and `floppy2` with a timeout of 10 minutes, whereas it was able to verify each of their respective PSS-CFGs in less than 30 seconds, thus providing a huge benefit to ARMC. For CPA-CHECKER, the benefit was relatively smaller, providing on average a speedup of 1.5. The reason is because CPA-CHECKER is a more sophisticated verifier than the other two, but still the fact that the PSS-CFG provides a speedup for CPA-CHECKER is to be considered noteworthy. Thus, we believe that the PSS-CFG is quite a useful object in general for verification.

### 3.12 Summary

In conclusion, we presented a new representation for programs with target variables. We started by considering the (full) symbolic execution tree in order to perform slicing on each path. The sliced paths remove both boolean guards and assignments, thus saving the symbolic execution process from some constraint solving. We overcame the main challenge which was to restrict attention to just the paths that exhibit dependencies toward the target variables. In the end, we showed that the blowup, in the final representation, is manageable. The most important result, however, was in the quality of the representation when it was used in a number of verification and testing benchmarks. In all experiments, the representation produced significant performance gains. Preliminary experiments of its use in other application areas are also promising.

# Chapter 4

## Concolic Testing

Testing is the most commonly used method to ensure software quality. It executes a given program with some inputs and the objective is then to find bugs or validate the program with respect to the given inputs. Traditionally, testing was carried out using manually generated inputs which became cumbersome and ineffective. Random testing alleviates this problem by generating random test inputs, but suffers from poor code coverage. Recently, more intelligent methods [97, 49, 18, 16] based on *concolic testing*, a variant of symbolic execution, have emerged, that generate inputs by systematically exploring program paths attempting to increase coverage.

The main idea of concolic testing is to execute the program simultaneously with concrete values and symbolic values. When the program is executed, symbolic constraints along the executed path are collected in a formula called a *path condition*. Then, a branch is picked and negated from the path condition resulting in a new formula which is then fed to a constraint solver to check for satisfiability. If it is satisfiable, concrete test inputs are generated to follow the new feasible path. If it is unsatisfiable, the new path is infeasible and another branch has to be picked to be negated. This way concolic testing attempts to improve the poor code coverage of random testing. A key characteristic of concolic testing is that path conditions can be simplified using concrete values whenever the decidability of their symbolic constraints goes beyond the capabilities of the underlying constraint solver.

One major problem with concolic testing is that there are in general an ex-

ponential number of paths in the program to explore, resulting in the so-called *path-explosion* problem. Recently, several methods have been proposed to attack this problem from various angles: using heuristics focused on branch coverage [16], function summaries [47], using static/dynamic program analysis [15] and so on. We propose a new method based on *interpolation*, largely complementary to existing approaches, that significantly mitigates path-explosion by pruning a potentially exponential number of paths that can be guaranteed to not encounter a bug.

Our method, inspired by [66], aims at assisting concolic testing by making use of the concept of *interpolation* [28] interleaved with the concolic execution process. The use of interpolation for pruning paths in the context of symbolic execution is well-known (see e.g., [66, 83]). The idea is as follows: first, assume that the program is annotated with certain bug conditions of the form “if  $C$  then **bug**”, where if the condition  $C$  evaluates to true along a path, the path is buggy. Then, whenever an unsatisfiable path condition is fed to the solver, an interpolant is generated at each program point along the path. The interpolant at a given program point can be seen as a formula that *succinctly* captures the reason of infeasibility of paths at the program point. In other words it succinctly captures the reason why paths through the program point are not buggy (infeasible paths are not buggy, by definition). As a result, if the program point is encountered again through a different path such that the interpolant is implied, the new path can be *subsumed*, because it can be guaranteed to not be buggy. The exponential savings are due to the fact that not only is the new path subsumed, but also the paths that this new path would spawn by negating its branches.

Unfortunately, methods such as [66, 83] cannot be used directly for concolic testing due to several challenges. First, the soundness of these methods relies on the assumption that an interpolant at a node has been computed after exploring the entire “tree” of paths that arise from the node. In concolic testing, this assumption is invalid as the tester can impose an arbitrary search order. For example, concolic testers such as Crest [16] and KLEE [17] use often many heuristics that may follow a random walk through the search space, thus making this method unsound. To address this problem, we need to keep track of nodes

whose trees have been explored fully (in which case we say the node is annotated with a *full-interpolant*) or partially (similarly, a *half-interpolant*).

Under this new setting, only nodes with full-interpolants are capable of subsumption in a sound manner. As a result, the amount of subsumption depends on how often nodes get annotated with full-interpolants from the paths explored by the concolic tester. Unfortunately our benchmarks in Section 4.5 showed that the above method by itself results in very few nodes with full-interpolants, thereby providing poor benefit to the concolic tester, because the tester rarely explores the entire tree of paths arising from a node. Hence, an important challenge now is to “accelerate” the formation of full-interpolants in order to increase subsumption. For this, we introduce a novel technique called *greedy confirmation* that performs limited path exploration (i.e., execution of a few extra paths) by itself, guided by subsumption, with an aim to produce a full-interpolant at nodes currently annotated with a half-interpolant. It is worth mentioning that this execution of few paths is done without interfering with the search order of the concolic tester. This technique resulted in a significant increase in subsumption for our benchmarks, and is vital for the effectiveness of our method.

We implemented our method and compared it with a publicly available concolic tester, Crest [16]. We found that for the price of a reasonable overhead to compute interpolants, a large percentage of paths executed by those heuristics can be subsumed thereby increasing their coverage substantially.

## 4.1 Related Work

The main innovation introduced by concolic testing (originally presented in DART [49] and Cute [97]) was the fact that concrete inputs can be generated based on some intelligent decision by symbolically negating one of the executed branches. Since the seminal papers of [49, 97] many works have been published improving concolic testing in different ways. We limit our discussion to related works that attempt at mitigating the scalability issues in concolic testing due to the exponential numbers of paths.

Recent extensions (e.g., EXE [18], Crest [16], Sage [51] and KLEE [17])

have tried to address this challenge by using novel heuristics to guide the exploration of paths improving the naive depth-first search strategy originally used in DART. They target *branch coverage* (i.e., number of branches evaluated to true and false) rather than *path coverage*. Although branch coverage does not suffer from path-explosion it is understood that the ultimate goal of the concolic tester is path coverage. Branch coverage is just an inexpensive measure of the quality of a test suite, and a good branch coverage is more of a necessary requirement for quality than sufficient. The main difference with the above methods is that we focus on path coverage rather than branch coverage while respecting those search strategies.

Another interesting line of research has addressed the caching of *function summaries* as a way of reducing the exponential number of paths (e.g., SMART [47] and [7]). Our method performs function inlining, thus being naive compared to them interprocedurally, while these methods execute the DART algorithm to generate reusable summaries, thus being naive compared to us intraprocedurally. This suggests that both approaches are orthogonal and could work together to benefit from each other.

The closest related tester to ours in the line of pruning paths is [15]. This method eliminates redundant paths by analysing the values read and written in memory by the program. The main idea is to prune paths that only differ in program variables that are not subsequently read (i.e., *dead variables*). We share with them the high-level idea of removing irrelevant information in order to increase the chances of subsumption, but the similarity ends there. The most important difference is that [15] defines “irrelevant information” using live (dead) variables so the subsumption test is simply a subset operation. We use interpolation to prune paths so our subsumption test involves logical entailment checks, which are more expensive. However, interpolants are much more powerful for subsumption, which can result often in a greater amount of pruned paths. We exemplify these differences with [15] through an example in Section 4.2.

Finally, as mentioned before, we have been clearly inspired by [66] in the idea of interpolating infeasible paths which has been also applied in [83]. However, [66, 83] assume full control of the search space by performing a DFS-

traversal to compute their interpolants, which ensures an interpolant at a node always represents the entire tree of paths below a node. This is the key to making sound subsumption. This assumption is no longer valid in concolic testing since the tester controls the search space using some heuristic which may not be DFS. Interestingly, Crest [16] compares different heuristics for concolic testing and concluded that DFS is actually the worst in terms of branch coverage. Thus, [66, 83] are not readily suitable for concolic testing. More importantly, as we will see in Section 4.5, even if somehow the DFS-restriction is lifted in these methods (e.g., by augmenting them with half and full-interpolants), it is not enough to provide a reasonable benefit to concolic testing, as they scale poorly without greedy confirmation.

## 4.2 Running Example

Consider the program in Fig. 4.1, where a `read()` call signals the concolic tester to generate a concrete input. In this case, the inputs are only used to decide the program’s control flow, and hence are not stored in a variable. Assume initially the concolic tester generates a positive value for both inputs. This drives the tester down the path  $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_6 \cdot \ell_8$ , shown by the left-most path in the program’s symbolic execution tree. A symbolic execution tree represents the set of paths traversed by the program where each node corresponds to a program location and an edge between two nodes corresponds to the program transition.

Now, the concolic tester provides us this path, on which we perform symbolic execution and annotate it with interpolants in a backward manner. At  $\ell_8$  the path is not buggy and there is no infeasibility, therefore the interpolant *true* is stored there (denoted by  $\ell_8 : \{true\}$ ). Propagating this to  $\ell_6$  we note that there is another branch (to  $\ell_7$ ) that has not been explored, so we annotate the interpolant *true* at  $\ell_6$  as a *half-interpolant* to denote this fact. Once a node has been annotated with a half-interpolant, we stop and give control back to the tester.

Assume now the concolic tester attempts to “turn-around” at  $\ell_6$  into the path  $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_6 \cdot \ell_7$ . This path is infeasible as its path condition  $s = 0 \wedge s' = s + 1 \wedge s'' = s' + 2 \wedge s'' > 3$  is unsatisfiable (for simplicity we omitted the `read`

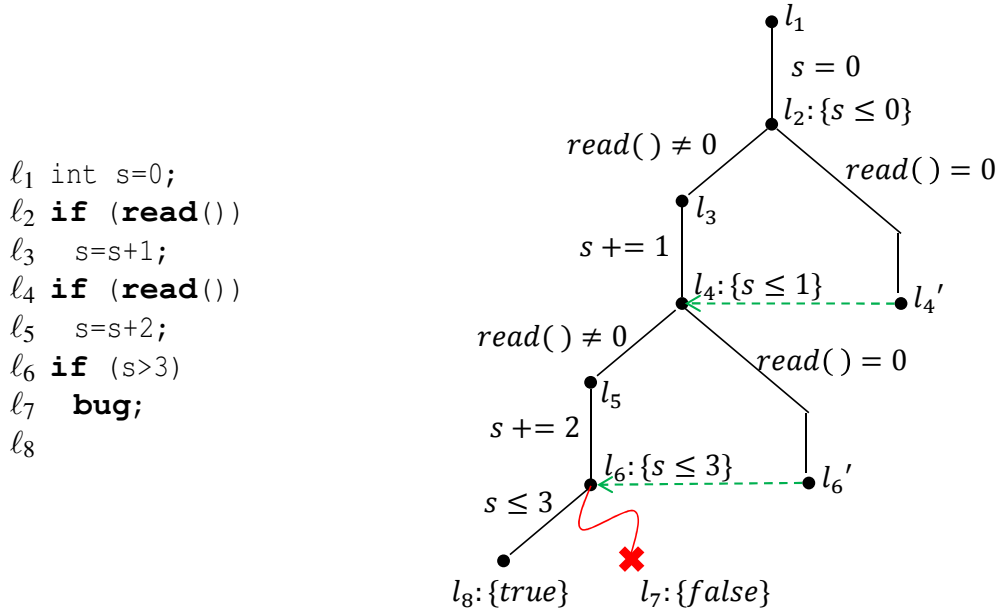


Figure 4.1: A program and its symbolic execution tree

constraints). Now we generate interpolants for this path by first annotating  $l_7$  :  $\{false\}$  because it is an unreachable node. Propagating this back to  $l_6$  through the label  $s > 3$ , we obtain the interpolant  $s \leq 3^1$ . We now note all paths from  $l_6$  have been explored, so we conjoin all interpolants at  $l_6$  ( $true \wedge s \leq 3$ ) annotating it with the *full-interpolant*  $s \leq 3$ , denoting that the entire tree of paths under  $l_6$  has been explored.

Now, when the concolic tester generates a zero for the second `read()` and executes the path  $l_1 \cdot l_2 \cdot l_3 \cdot l_4 \cdot l_6'$ , we check<sup>2</sup> if the path condition at  $l_6'$ ,  $s = 0 \wedge s' = s + 1$  implies the full-interpolant at  $l_6$ ,  $s' \leq 3$  (after proper renaming). It does, so we can guarantee the entire tree of paths below  $l_6'$  to not be buggy and *subsume* it. The exponential savings is because we saved the concolic tester from executing a potentially exponential number of paths in the tree under  $l_6'$  (in this case, two paths, but in general exponential). Importantly, note that only full-interpolants are capable of subsumption and the amount of savings provided by our method is directly proportional to the number of full-interpolants formed from exploring entire trees of paths.

Unfortunately, the method discussed so far has a catch. We conveniently

<sup>1</sup>Note that for this example we use interpolants based on weakest preconditions, but our method is not limited to them and any interpolation method can be used.

<sup>2</sup>The symbolic path in fact continues to  $l_8$ , but our check succeeds and stops execution at  $l_6$ .



assumed the concolic tester would execute the paths in that specific order, which resembles a depth-first search (DFS). If after executing the first path the tester did not attempt to turn-around at  $\ell_6$  (thus leaving  $\ell_6$  with a half-interpolant),  $\ell_6$  would not have been able to subsume  $\ell'_6$ . Even worse is the fact that because of the half-interpolant at  $\ell_6$  all its ancestors along the path also become incapable of subsumption, thereby losing a great amount of savings.

Thus, we need to “accelerate” the formation of full-interpolants instead of simply relying on the concolic tester to explore the paths. However a challenge in concolic testing is that whatever technique is employed for this purpose, it must always stay *proportional* to the executed path, that is, not become intractable. For this, we introduce a technique called *greedy confirmation* that is both tractable (proportional to the path length in the worst-case) and accelerates the formation of full-interpolants resulting in a substantial increase in subsumption.

The basic idea is while backtracking along a path, once we encounter a node with another branch that has not been explored by the concolic tester, instead of simply annotating the node with a half-interpolant and halting the process, we explore the other branch ourselves (while the concolic tester is waiting to generate the next path) and attempt to confirm whether a full-interpolant can be generated from it, so that we can annotate the node as full and continue the backward propagation. However, the sub-tree under the other branch could be exponentially large, in which case we must avoid exploring it to remain tractable. Hence we introduce a restriction: while exploring the sub-tree, each program point is allowed to be explored at most once. If a program point is visited along more than one path, we demand that it be subsumed. If not, we declare that the greedy confirmation process failed at the original branch node, which will remain annotated with a half-interpolant.

The reasoning behind the restriction is as follows. During greedy confirmation, we want to give each program point at least one chance to become subsumed, so we must allow at least one visit to each program point. However, we do not want to resort to a full search within the sub-tree, which could become intractable. Thus, this restriction ensures that the “search” is linearly bounded

by the longest path in the program<sup>3</sup>. The motivation behind this technique is that the difference between the two branching sub-trees might not affect the bug condition and so one tree can quickly subsume the other.

Let us see how greedy confirmation works on the example. When back-tracking along the first path,  $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_6 \cdot \ell_8$ , once we reach  $\ell_6$  with the interpolant *true*, we trigger greedy confirmation and attempt to explore the other tree under  $\ell_6$ . We immediately notice the other branch to  $\ell_7$  is infeasible, and annotate it with *false*. Propagating this back to  $\ell_6$  we get the *full*-interpolant  $s \leq 3$ , which can now be propagated back further. Hence at  $\ell_5$  we get the full-interpolant  $s \leq 1$ . Propagating this to  $\ell_4$  makes it a half-interpolant since there is another branch from  $\ell_4$  to  $\ell'_6$ . Triggering greedy confirmation, this time at  $\ell_4$ , we notice that  $\ell'_6$  is subsumed by  $\ell_6$  with the full-interpolant  $s \leq 3$ . Propagating this to  $\ell_4$  and conjoining it with the existing half-interpolant, we get  $s \leq 3 \wedge s \leq 1$ , or simply  $s \leq 1$  at  $\ell_4$ , which is now a *full*-interpolant that can be propagated back. Similar reasoning at  $\ell_2$  subsumes a large tree of paths under  $\ell'_4$  resulting in the full-interpolant  $s \leq 0$  at  $\ell_2$ .

Now, in this contrived example, if the concolic tester executes any path in the program, we are able to subsume it immediately at  $\ell_2$ . Essentially, greedy confirmation has pushed subsumption from happening at lower levels in the symbolic execution tree to upper levels. A simple but effective optimisation is if greedy confirmation failed at a node, thereby leaving it with a half-interpolant, we can simply halt the annotation process because there is no use propagating a half-interpolant to the node's parents.

Finally, note importantly that this example does not contain any dead variables, so techniques like [15] will not be able to prune any path, whereas with interpolation we are able to.

**Remark.** The correctness of our method relies on the fact that whenever we subsume a path (i.e., we skip its execution) we can ensure that the path will not be buggy (see Theorem 4 in Sec. 4.4). This guarantee is only feasible if the program is annotated with assertions. Without them we have no basis to make such a guarantee and subsume the path. A key observation is that without

---

<sup>3</sup>We tried restricting to two, three and other constant instances of each program point, but we found no difference in our benchmarks.

assertions each path is unique because at least it will differ in one branch from the rest of paths, and hence, there is no hope for boosting the concolic execution. However, with an assertion many paths can be considered equivalent and this allows our method reporting savings. In conclusion, our method is only effective if we consider programs annotated with assertions. We believe that this step can be done automatically and it is not a big hassle in practice. Of course, our technique works with multiple assertions in the program.

### 4.3 Background

**Concolic Testing.** We first define a *concolic path*  $p$  as the path executed by the concolic tester represented as a sequence of program locations  $\ell_1 \cdot \dots \cdot \ell_n$  and define the transition relation  $\ell \xrightarrow{\text{op}} \ell'$  as before. In order to manipulate concolic paths we also define  $\text{prefix}(p, \ell)$  of a path  $p$  w.r.t. a location  $\ell$  as the prefix up to  $\ell$  without including  $\ell$ . We also define  $\text{constraints}(p)$  that maps a concolic path into a formula representing the conjunction of the symbolic constraints along the path. Of course, this formula is properly renamed (i.e., SSA form) to take into account variables that are redefined more than once.

We now show in Fig. 4.2 a generic algorithm that performs concolic testing as described in [16]. The algorithm is generic in the sense that it can be parameterised with different search strategies by simply choosing different heuristics to pick a branch at line 2. The algorithm is instantiated with the program  $P$  and an initial concolic path  $p$ , usually chosen by running the program with random inputs. In line 2, a branch  $b_i$  is chosen from the path  $p$  based on the heuristic used. In line 3, a new path  $p'$  is built by keeping the prefix up to  $b_i$  and negating the constraints at the branch  $b_i$ . In line 4 the symbolic constraints associated with  $p'$  are then fed to a constraint solver to check for satisfiability. If they are unsatisfiable, the algorithm returns to line 2 and picks another branch to negate. Otherwise, an assignment  $I$  (concrete inputs) is extracted from the solver which is used to guide the next concrete path along the negated branch. This is done using the call to  $\text{ConcretePath}(P, I)$  in line 5. Once the new path, say  $q$ , is executed, depending on the heuristic, it is processed by either replacing the old path

```

    GENERICCONCOLIC(program  $P$ , path  $p$ )
1: while termination conditions are not met do
2:    $b_i$  = pick a branch from  $p$ 
3:    $p'$  = construct a path passing through
         the branches  $b_0, \dots, b_{i-1}, \bar{b}_i$ 
4:   if  $\exists$  satisfying assignment  $I$  forcing  $P$  through  $p'$  then
5:      $q \leftarrow \text{ConcretePath}(P, I)$ 
6:     process  $q$  by either  $p \leftarrow q$  or
         GENERICCONCOLIC( $P, q$ )
7:   endif
8: endwhile

```

Figure 4.2: A Generic Concolic Tester

$p$  with  $q$  or by making a recursive call to GENERICCONCOLIC with  $q$  in line 6. The entire process continues until some termination condition is met (usually a fixed number of iterations or recursive call depth) shown in line 1.

## 4.4 Algorithm

We now present our symbolic execution based algorithm that runs synchronised with GENERICCONCOLIC and helps the concolic testing strategy mitigate the path-explosion problem. First, we give few definitions critical to our algorithm:

**Definition 8** (Full and half interpolants). *An interpolant annotated at a symbolic state  $v$  is a full-interpolant if either:*

- (a)  $v$  is a leaf node (terminal, infeasible, subsumed) in the symbolic tree, or
- (b)  $\forall v'$  such that  $\text{SUCC}(v, v')$ ,  $v'$  is annotated with a full-interpolant.

*An interpolant that is not full is called a half-interpolant<sup>4</sup>.*

**Definition 9** (Subsumption check). *Given a current symbolic state  $v \equiv \langle \ell, s, \cdot \rangle$  and an already explored symbolic state  $v' \equiv \langle \ell, \cdot, \cdot \rangle$  annotated with the interpolant  $\bar{\Psi}$ , we say  $v$  is subsumed by  $v'$  ( $\text{SUBSUME}(v, \langle v', \bar{\Psi} \rangle)$ ) if  $\bar{\Psi}$  is a full-interpolant and  $\llbracket v \rrbracket_s \models \bar{\Psi}$ .*

The subsumption check here is defined similar to the notion of merging defined in Definition 3, but with two differences. Firstly, in the setting of concolic

<sup>4</sup>“Partial-interpolant” is a better term but we use half-interpolant for historical reasons.

```

SYMEXEC( $v \equiv \langle \ell, \cdot, \cdot \rangle, p$ )
1:  if TERMINAL( $v$ ) then  $\langle \bar{\Psi}, f \rangle = \langle true, full \rangle$ 
2:  else if INFEASIBLE( $v$ ) then  $\langle \bar{\Psi}, f \rangle = \langle false, full \rangle$ 
3:  else if  $\exists v' \equiv \langle \ell, \cdot, \cdot \rangle$  annotated with  $\langle \bar{\Psi}', full \rangle$  such that SUBSUME( $v, \langle v', \bar{\Psi}' \rangle$ ) then
4:     $\langle \bar{\Psi}, f \rangle = \langle \bar{\Psi}', full \rangle$ 
5:  else  $\langle \bar{\Psi}, f \rangle = \text{UNWINDPATH}(v, p)$ 
6:  endif
7:  annotate  $v$  with  $\langle \bar{\Psi}, f \rangle$ 
8:  if  $f \neq full$  then halt

UNWINDPATH( $v \equiv \langle \ell, s, \Pi \rangle, p$ )
1:   $\bar{\Psi} =$ existing interpolant at  $v$ 

2:   $v' \triangleq \begin{cases} \langle \ell', s, \Pi \wedge \llbracket c \rrbracket_s \rangle & \text{if } \ell \xrightarrow{\text{op}} \ell' \in p \wedge \text{op} \equiv \text{assume}(c) \\ \langle \ell', s[x \mapsto S_x], \Pi \rangle & \text{if } \ell \xrightarrow{\text{op}} \ell' \in p \wedge \text{op} \equiv x = e \text{ and } S_x \text{ fresh variable} \end{cases}$ 

3:  SYMEXEC( $v', p$ ) and let the resulting annotation at  $v'$  be  $\langle \bar{\Psi}', \cdot \rangle$ 
4:   $\bar{\Psi} = \bar{\Psi} \wedge \text{INTP}(\text{constraints}(\text{prefix}(p, \ell)), \text{constraints}(\ell \cdot \ell') \wedge \neg \bar{\Psi}')$ 
5:  if  $\exists v'' \equiv \langle \ell'', \cdot, \cdot \rangle$  such that  $v''$  is not annotated  $\langle \cdot, full \rangle$  and SUCC( $v, v''$ ) then
6:    GREEDYCONFIRMATION( $v''$ ) and let the resulting annotation at  $v''$  be  $\langle \bar{\Psi}'', f'' \rangle$ 
7:    if  $f'' = full$  then
8:       $\bar{\Psi} = \bar{\Psi} \wedge \text{INTP}(\text{constraints}(\text{prefix}(p, \ell) \cdot \ell), \text{constraints}(\ell \cdot \ell'') \wedge \neg \bar{\Psi}'')$ 
9:    endif
10:  endif

11:   $f \equiv \begin{cases} full & \text{if } \forall v''' \text{ s.t. } \text{SUCC}(v, v'''), v''' \text{ is annotated with } \langle \cdot, full \rangle \\ half & \text{otherwise} \end{cases}$ 

12:  return  $\langle \bar{\Psi}, f \rangle$ 

```

Figure 4.3: Symbolic execution with interpolation along a path

testing, there needs to exist a distinction between full and half interpolants. Note that a half-interpolant at a symbolic state cannot be used to subsume other states. Secondly, the subsumption check here does not consider witnesses due to the reason mentioned in Section 2.2.

#### 4.4.1 Symbolic Execution of Paths with Interpolants

We first present the algorithm SYMEXEC (Figure 4.3) that takes a path chosen by the concolic tester and executes it symbolically in order to annotate each program point in it with interpolants. A collection of such annotated paths implicitly represents the symbolic execution tree. Then, we will present GENER-

ICCONCOLICWITHPRUNING, a modified version of GENERICCONCOLIC that can prune paths using the annotated symbolic execution tree.

SYMEXEC takes as arguments a symbolic state  $\nu$  and a path  $p$ . Initially,  $\nu$  refers to the start state  $\langle \ell_{\text{start}}, \cdot, \cdot \rangle$ . It is also assumed that all procedures have access to the program's original transition system  $P$ , which can be considered a global variable to the algorithm. The actual object of interest is the annotation done by the procedure which is persistent across multiple calls to it. For the sake of simplicity, ignore the gray box which we will come to later.

First, lines 1-4 handle the three possible base cases for the symbolic execution of a path: terminal, infeasible and subsumed. In line 1, the function TERMINAL checks if  $\ell = \ell_{\text{end}}$ . If yes, the path can be fully generalised returning the full-interpolant *true*, since it is feasible. In line 2, the function INFEASIBLE checks if the path condition  $\Pi$  of  $\nu$  is unsatisfiable. If yes, again the path can be fully generalised, but this time to *false* since it is infeasible. Finally, line 3 checks if there is another state  $\nu'$  that can subsume  $\nu$ , using the function SUBSUME which implements Definition 9. If yes, we can simply annotate  $\nu$  with the (full) interpolant of  $\nu'$  (line 4).

If the three base cases described above are not applicable, the algorithm executes symbolically the next location of the path by calling the procedure UNWINDPATH. This procedure, at line 1, obtains any interpolant that may be annotated at  $\nu$  (it can be assumed that initially all symbolic states are annotated with the default interpolant  $\langle \textit{true}, \textit{half} \rangle$ ). In line 2, it executes one symbolic step along the path and calls the main procedure SYMEXEC with the new successor state  $\nu'$  (line 3). This mutually recursive call will in the end annotate  $\nu'$  with an interpolant, say  $\bar{\Psi}'$ . In line 4, it computes the interpolant for the current state  $\nu$ , using  $\bar{\Psi}'$  and the constraints along the transition  $\ell \xrightarrow{\text{op}} \ell'$  and then conjoining the result with any existing interpolant at  $\nu$ . Finally, in line 11, it computes whether  $\bar{\Psi}$  is a full or half-interpolant by implementing Definition 8. For example, consider the case where  $\textit{constraints}(\textit{prefix}(p, \ell)) \equiv x_0 = 1 \wedge x_1 = x_0 + 3$ ,  $\textit{constraints}(\ell \cdot \ell') \equiv x_2 = x_1 + 2$ , and  $\bar{\Psi}' = x_2 \leq 10$  (after proper renaming). The call at line 11 will generate an interpolant whose variables must be common to  $\textit{constraints}(\textit{prefix}(p, \ell))$  and  $\textit{constraints}(\ell \cdot \ell')$  together with  $\bar{\Psi}'$ . Thus, it can

```

    GENERICCONCOLICWITHPRUNING(program  $P$ , path  $p$ )
1: while termination conditions are not met do
2:    $b_i$  = pick a branch from  $p$ 
3:    $p'$  = construct a path passing through
         the branches  $b_0, \dots, b_{i-1}, \bar{b}_i$ 
4:   if ISSUBSUMEDPATH( $p'$ ) then continue
5:   endif
6:   if  $\exists$  satisfying assignment  $I$  forcing  $P$  through  $p'$  then
7:      $q \leftarrow$  ConcretePath( $P, I$ )
8:     SYMEXEC( $v_0 \equiv \langle \ell_{\text{start}}, \text{init\_store}(), \text{true} \rangle, q$ )
9:     process  $q$  by either  $p \leftarrow q$  or
         GENERICCONCOLICWITHPRUNING( $P, q$ )
10:  endif
11:endwhile

```

Figure 4.4: A Generic Concolic Tester with Pruning

only include  $x_1$ . An interpolant generated by MathSAT [23] would be  $x_1 \leq 4$ . A weaker interpolant is  $x_1 \leq 8$  which can be computed by weakest preconditions.

Returning back the tuple  $\langle \bar{\Psi}, f \rangle$  to the caller SYMEXEC, at line 7, it performs the actual annotation of  $v$ . Finally, in line 8, SYMEXEC checks if it just annotated the state with a full interpolant. If not, it halts in a normal manner and returns control to the concolic tester (this can be seen as a system-wide halt and is also notified to UNWINDPATH)<sup>5</sup>. The reason for halting is that there is no use propagating the half interpolants to parent nodes, because they will still remain as half interpolants and be of no use for subsumption. In other words, the algorithm only propagates full interpolants to parent nodes. This is a simple but very effective optimisation. Note however, that the annotation of states done so far is still persistent. More importantly, a half interpolant at a node now could get converted to a full interpolant later when all successors of the node get full interpolants.

We now present a modified concolic tester GENERICCONCOLICWITHPRUNING, shown in Fig. 4.4, with the gray boxes highlighting the changes made in order to prune paths using our method. First, after picking a branch  $b_i$  to negate (line 2) and constructing the corresponding path  $p'$  that goes through

<sup>5</sup>This halting may make the algorithm seem “impure” but we believe it makes it much easier to understand.

it (line 3), the concolic tester makes a call to `ISSUBSUMEDPATH` with  $p'$  at line 4. This call queries the persistent annotated symbolic execution tree to check whether the path  $p'$  has been subsumed already. Note that this is just a look-up and hence does not involve symbolic execution. If yes, the tester can simply skip  $p'$  from being executed and continue, thus pruning a potentially exponential number of recursive calls at line 9 and all those paths it would have generated. The second change is that once a path has been *concolically* executed in line 7, the tester has to invoke our method so that we can annotate it with interpolants. Thus, line 8 makes a call to `SYMEXEC` with the initial state of the path  $v_0$  and the path  $q$  that was executed.

#### 4.4.2 Greedy Confirmation to Accelerate Formation of Full-Interpolants

The method discussed so far has laid out the framework to bring interpolation to concolic testing with the help of half and full-interpolants. The benefit provided by our technique relies heavily on the formation of full-interpolants from paths explored by the concolic tester (i.e., the amount of times `f` is assigned full in `UNWINDPATH`, line 11). Unfortunately the method explained so far does not perform well in practice (as we will see in Section 4.5) because relying only on the tester to explore paths results in poor formation of full-interpolants. This is because the tester’s arbitrary search strategy seldom explores the entire tree of paths arising from a node, a requirement to annotate the node with a full-interpolant and enable it to subsume other nodes.

Hence, an important challenge to deal with in concolic testing is to “accelerate” the formation of full-interpolants. For this, we introduce a new concept called *greedy confirmation*. The basic idea was described in Section 4.2, here we explain it in technical terms (gray box in Fig. 4.3). If the call to `SYMEXEC` (line 3 of `UNWINDPATH`) returned successfully, it means  $v'$ , the successor of  $v$ , was annotated with a full-interpolant, say  $\bar{\Psi}'$ . Now, we check (at line 5) if the other successor of  $v$  (say,  $v''$ ) is also annotated with a full-interpolant. If not, then this half-interpolant, say  $\bar{\Psi}''$ , is the one preventing  $\bar{\Psi}$  from becoming a full-interpolant. Now, we greedily explore  $v''$  by ourselves in an attempt to



confirm whether we can make  $\overline{\Psi}''$  a full-interpolant, so that we can immediately upgrade  $\overline{\Psi}$  to full, thus enabling it to perform subsumption. The intuition behind this technique is that the difference between the trees below the two “siblings”  $v'$  and  $v''$  might have no effect on the bug condition, and since the tree below  $v'$  has been fully explored, thereby annotated with full-interpolants, it opens up opportunities to subsume nodes in the tree below  $v''$ . Thus, we make the call to `GREEDYCONFIRMATION` in line 6.

`GREEDYCONFIRMATION` essentially performs symbolic execution of  $v''$  similar to `SYMEXEC`. However, a key impediment is that the symbolic execution of  $v''$  must not become expensive and it should have a *proportional* cost to the length of path, otherwise we consider it intractable. However the tree under  $v''$  might be arbitrarily large and so we impose an important restriction to maintain tractability: each program point is allowed to be explored at most once during greedy confirmation of  $v''$ . If a program point  $\ell_k$  is visited in two states  $v_{k_1}$  and  $v_{k_2}$  then we demand that one of them be subsumed (not necessarily by the other). If it is not possible, then we declare that the invocation of greedy confirmation failed at  $v''$ , which will not be annotated with a full-interpolant. This restriction ensures that in the worst case, greedy confirmation at any symbolic state is bounded linearly by the length of the longest path in the program.

Note that `GREEDYCONFIRMATION` does not need the help of the concolic tester to explore its paths. In fact, this is the whole point. Thus, it does not take as argument the path  $p$ , but instead explores paths on its own. Since `GREEDYCONFIRMATION` is almost exactly identical to `SYMEXEC`, we do not show its (rather tedious) pseudo-code here. The only modification to `SYMEXEC` is to keep track of counters for each visited program location that is not subsumed and stop if a counter becomes greater than one. Finally, to perform the symbolic step (line 3, `UNWINDPATH`), `GREEDYCONFIRMATION` must pick  $\ell \xrightarrow{\text{op}} \ell'$  from the program’s transition system  $P$  instead of the path  $p$ .

Coming back to the description of `UNWINDPATH`, consider the call to `GREEDYCONFIRMATION` at line 6 produced the annotation  $\langle \overline{\Psi}'', f'' \rangle$  at  $v''$ . If suppose  $\overline{\Psi}''$  was a full-interpolant (i.e., greedy confirmation succeeded), then at line 8 we augment the interpolant  $\overline{\Psi}$  at  $v$  with this full-interpolant. More im-

portantly, at line 11,  $f$  will be assigned full because both successors of  $v$  ( $v'$  and  $v''$ ) have been annotated with full-interpolants ( $\bar{\Psi}'$  and  $\bar{\Psi}''$  respectively). This directly accelerates the formation of full-interpolants, resulting in more subsumption (line 3 of SYMEXEC succeeding more often). In Section 4.5 we will see this greatly increases the savings provided for concolic testing, while still maintaining tractability.

**Theorem 4.** *When a symbolic state  $v$  is subsumed by another state  $v'$ , the error location  $\ell_{\text{error}}$  is unreachable from  $v$ .*

The proof follows from the correctness of the algorithms described in [66, 83], Lemma 1, and the fact we only subsume if the interpolant is full.

**Corollary 5.** *If GENERICCONCOLIC will execute a path  $p$  that leads to the bug location  $\ell_{\text{error}}$ , then GENERICCONCOLICWITHPRUNING will also execute  $p$ .*

Theorem 4 states that we never subsume a node from which the error location  $\ell_{\text{error}}$  would be reachable. Thus, it can be shown that we never prevent GENERICCONCOLICWITHPRUNING from executing a path that will reach  $\ell_{\text{error}}$ .

Finally, note that it is entirely possible that the concolic tester can detect infeasible paths after simplifying a complex constraint using concrete values that we cannot detect symbolically. Therefore if unsatisfiability cannot be determined symbolically, we cannot compute interpolants. As a result, the pruning of paths is limited only to those where unsatisfiability does not require reasoning about complex constraints. Moreover, it is possible for GREEDYCONFIRMATION to reach  $\ell_{\text{error}}$  during the exploration of a path in the presence of constraints whose unsatisfiability cannot be determined. In this case, we also declare that greedy confirmation failed and return the control to the concolic tester, since only the tester can decide the reachability of  $\ell_{\text{error}}$  in order to ensure zero tolerance for false alarms.

## 4.5 Experimental Evaluation

We implemented our algorithm on the TRACER [64] framework for symbolic execution and modified the concolic tester CREST [16] to communicate with

TRACER during its testing process. To achieve this, the algorithm in Fig. 4.3 was implemented in TRACER while CREST was slightly modified to implement the procedure in Fig. 4.4. The two work as separate processes coordinating their tasks.

We conducted experiments on three strategies: Depth-First Search (DFS), Uniform Random Search (URS) and Control Flow Graph (CFG) directed search. DFS explores the paths in a depth-first order naturally forming full-interpolants in a bottom-up manner (even without the need for greedy confirmation) and maximising the benefit of subsumption, so it is the best-case strategy for our method. However according to [16], it is the worst strategy in terms of branch coverage, so we mainly provide it for completeness. CFG is a heuristic-based strategy that first computes the shortest distances between every basic block using the CFG of the program and decides on a “target” basic block to cover. Then, when a path is executed it chooses the next path by turning around at a branch along the path with the *least distance* to the target block. If it is unable to do so (due to infeasibility of constraints), it chooses the next closest branch to the target that is along the path, and so on. Once the target is reached, it randomly chooses a new target block to cover. It is worth noting that CFG-directed search was shown to be best strategy in terms of branch coverage by [16], so we consider it an illustrative concolic testing strategy. Finally, URS explores paths in a random order by choosing to turn-around at a random branch in a currently executed path. We included URS just to provide another example of a typical random concolic testing strategy. This random element in both CFG and URS hinders greatly with the formation of full-interpolants and is the main challenge for our algorithm to overcome.

As benchmarks, we used four programs from the ntdrivers-simplified category of SV-COMP’12: cdaudio, diskperf, floppy and kbfiltr. These programs range from 1000 to 2000 lines of code. In spite of their relative small sizes in terms of LOC, these programs have a large number of paths due to the number of branches. Thus, we believe these programs can stress more the computation of interpolants and subsumption checks since symbolic paths contain a high number of constraints and it is often harder to reason about the infeasibility of

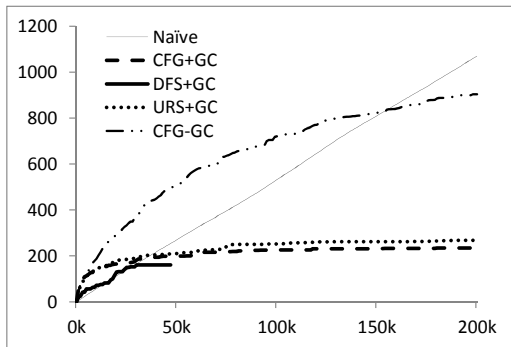
these constraints. One of the consequences is that interpolants are stronger (in the logical sense) since they must consider multiple reasons of infeasibility and hence, they are less likely to be reusable. We ran three main experiments with them on an Intel 3.2Ghz with 2Gb memory.

A technical problem with the experiments is that if our algorithm prevented CREST from exploring subsumed trees, we would never know how many paths CREST *would have* executed in those trees (and subsequently the time taken for the same) had we not prevented it. Also, we do not want to meddle with the random number-based sequence in CFG and URS by forbidding CREST to execute certain paths. Hence for measurement purposes, we do not forbid CREST from exploring subsumed trees in our experiments, but instead take note when CREST is in subsumed trees and discard any measurements taken during that time.

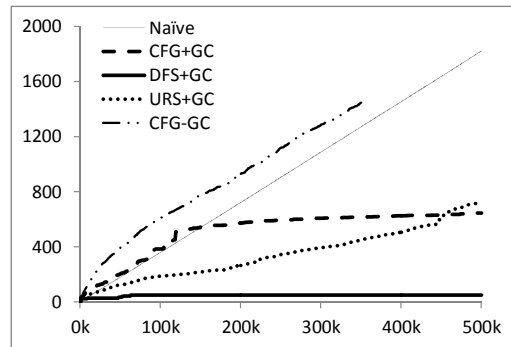
## First experiment – Performance

At the outset we would like to measure the actual performance improvement provided by our method. For this we measure the time taken by each strategy of naive CREST (i.e., original version) and CREST aided by our method (i.e., algorithms in Figs. 4.3 and 4.4) to execute a target number of paths, chosen based on the size of the benchmark: 200k for cdaudio and floppy, 500k for diskperf and around 20k for kbfiltr (due to its smaller size). The results are shown in Fig. 4.5, where the number of executed paths is shown on the X-axis and the time taken (in seconds) to execute those paths is shown on the Y-axis. For naive CREST, we noticed that the time taken to execute a path is almost constant across all three strategies, so for simplicity we took the average time of the three strategies (shown as Naive in Fig. 4.5). When running with our method (and with greedy confirmation), we show the strategies separately labelled as CFG+GC, DFS+GC and URS+GC.

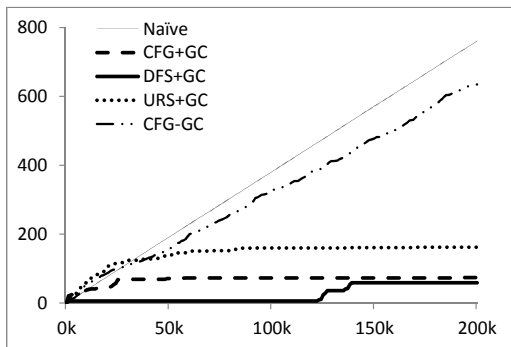
It can be seen that in each benchmark, naive CREST takes almost linear time to execute the target number of paths because the generation of each path involves simply one constraint solving, which takes almost constant time, and possibly a heuristic to choose the branch to negate, which is negligible. When aided by our method, CREST starts out a bit slower due to the overhead of in-



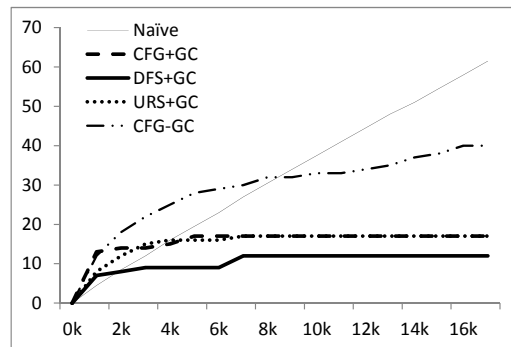
(a)



(b)



(c)



(d)

Figure 4.5: Timing for (a) cdaudio (b) diskperf (c) floppy (d) kbfiltr. X-axis: Paths, Y-axis: time in seconds

terpolation, however after a certain time the benefits of interpolation (i.e., sub-sampling) start to pay off, ultimately reaching the target number of paths much faster. The magnitude of improvement depends on the strategy. As expected, our method works best with DFS, providing more than 10-20 times improvement, followed by CFG and URS with about 3-5 times improvement. For example, in diskperf, naive CREST takes on average 1700s to complete, whereas with our aid, CFG and URS take about 600s and DFS takes 50s. Note that every path subsumed in this experiment is a path executed by CREST that need not have been executed. More importantly, the *trend* of the graphs of CREST running with our algorithm indicates potentially exponential benefit in time over naive

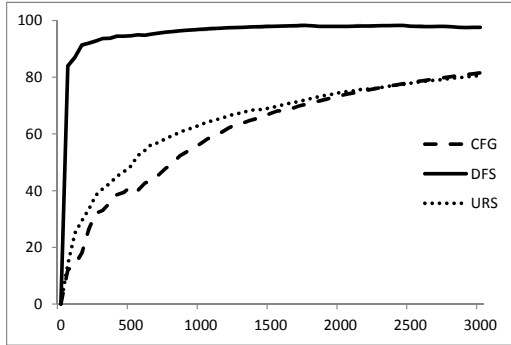
CREST.

To measure the effectiveness of greedy confirmation in our algorithm, we measured the timing of CREST with our algorithm but with greedy confirmation turned off (i.e., the gray box in Fig. 4.3 removed). Hence, the full-interpolants would be generated only by CREST exploring full sub-trees in the symbolic execution tree. We chose the canonical CFG strategy for this experiment, and its result is shown as CFG-GC (double-dotted line) in the graphs in Fig. 4.5. It can be seen immediately that the timing without greedy confirmation is much worse than otherwise. Especially for `diskperf` in Fig. 4.5(b), the timing is even slower than running naive CREST, because the overhead incurred due to interpolation does not pay off in subsuming other paths. In other benchmarks it pays off resulting in a benefit to CREST, but the amount of benefit is small compared to running CFG with greedy confirmation (CFG+GC). More importantly, the trend of CFG-GC does not appear to provide exponential benefit to CREST, if at all. This experiment shows that greedy confirmation is a indeed major contribution to the effectiveness of our algorithm and interpolation-based methods without it [83, 66] are not readily suitable for concolic testing.

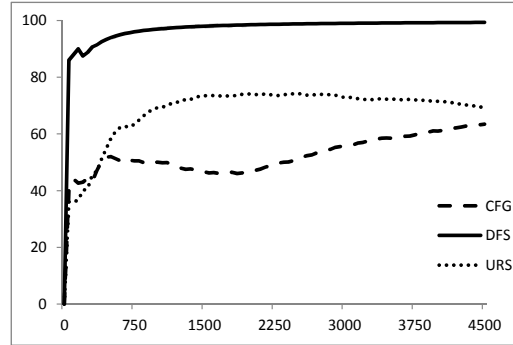
## Second experiment – Subsumption

Now we would like to understand the basis on which the first experiment provided benefit. A good measure for this is the amount of subsumption that our method provides, i.e., the percentage of CREST paths that are subsumed. A thing to take note is that CFG and URS may repeat execution of some paths due to their random element which we cannot control. For this and the subsequent experiment, we do not include such paths in the calculations because a repeated path does not contribute to the percentage of subsumption. We included them in the previous experiment because they indeed contribute to execution time.

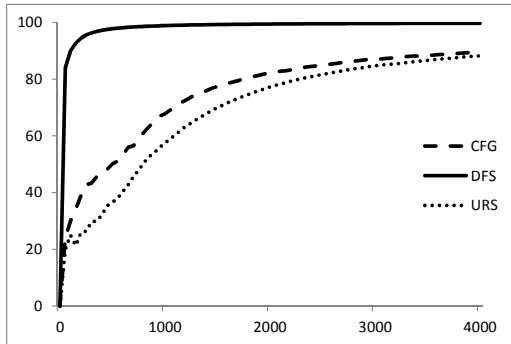
The results are shown in Fig. 4.6. In the graph of each benchmark, the X-axis represents the number of paths executed by CREST and the Y-axis represents the percentage of those paths subsumed. It can be clearly seen that DFS very quickly reaches almost 100% subsumption (i.e., almost all executed paths are subsumed by interpolants) whereas the other two strategies CFG and URS show



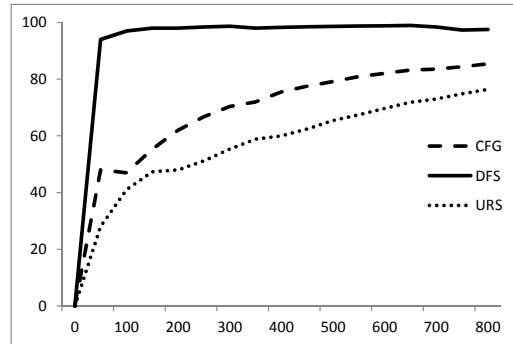
(a)



(b)



(c)



(d)

Figure 4.6: Subsumption for (a) cdaudio (b) diskperf (c) floppy (d) kbfiltr. X-axis: Paths, Y-axis: % subsumption

an increasing trend towards it, with DFS as their asymptote. This is in line with the first experiment where the magnitude of performance benefit was the greatest in DFS, followed by the other two. In diskperf we notice some noise, as CFG and URS fluctuate between 50-65% subsumption. The next experiment below suggests that this is because even though we subsume trees during this period, CFG and URS are not interested in executing paths in those trees. This is possibly why the magnitude of improvement was low (only around 2) for diskperf in Fig. 4.5(b). Theoretically, the rate of subsumption could decrease (even be zero) if CREST constantly avoids executing paths in subsumed trees, but we expect this to seldom happen in reality.

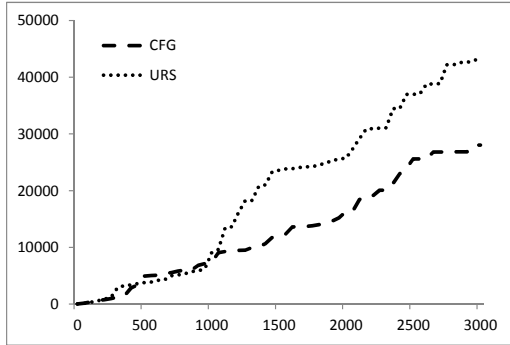
### Third experiment – Extra path-coverage

Now we present a different view of the provided benefit. When we subsume a tree, we not only provide the direct benefit of saving the paths that CREST indeed executes in the tree (the first experiment), but also the indirect benefit of covering paths in the tree that CREST cannot cover within its budget. Although such paths are of low priority to the strategy’s heuristics, they are provided “free of charge” by our method because the time we spend to subsume the whole tree is inclusive of these paths as well. This can be considered extra path coverage because CREST has no hope of executing them in its budget, but if its budget were longer it may execute them in future. Note that this experiment does not make sense for DFS, because when we subsume a tree, DFS would immediately proceed to execute *all* paths in the tree anyway.

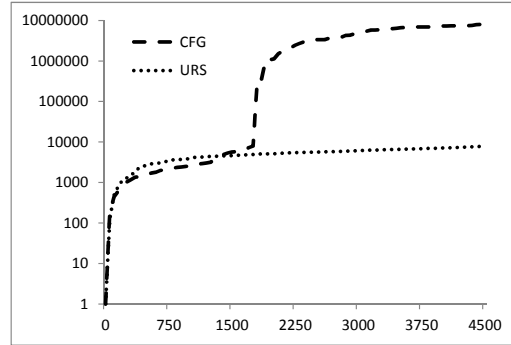
In Fig. 4.7, we measure the number of such “free” paths per path executed by CREST. Note that we again do not include repeated paths in this experiment since they do not contribute to path coverage. In each graph, the X-axis shows the actual path coverage of CREST, and the Y-axis shows the extra path coverage obtained due to subsumption. The ratio of extra path coverage to CREST’s actual path coverage varies greatly depending on the benchmark, from a magnitude of 10 in *cdaudio* to about 1000 in *diskperf* (note the logarithmic scale). Specifically, for the CFG strategy in *diskperf*, we subsume a huge number of trees around 1500 paths, but the previous experiment showed yet fluctuating percentage of subsumption around that time, indicating that CFG is not interested to execute paths in those subsumed trees.

In this experiment, by “taking credit” for entire subsumed trees, we measured the upper bound on the magnitude of benefit in path coverage that we can provide, a mean of 100. The lower bound, around 3 to 5, is dictated by the first experiment (although we did not explicitly measure it there, we can extract it from the timing), where we took credit only for the paths that CREST actually executed in the subsumed trees, within its budget. In general, one can expect the benefit we provide to lie somewhere in between, depending on the budget.

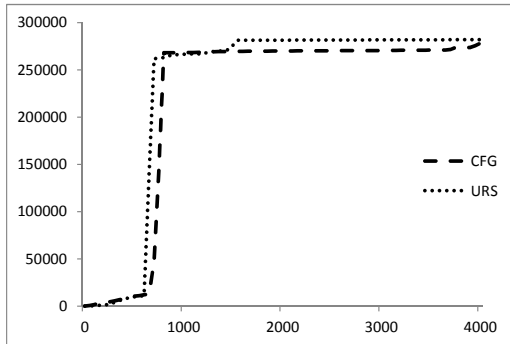




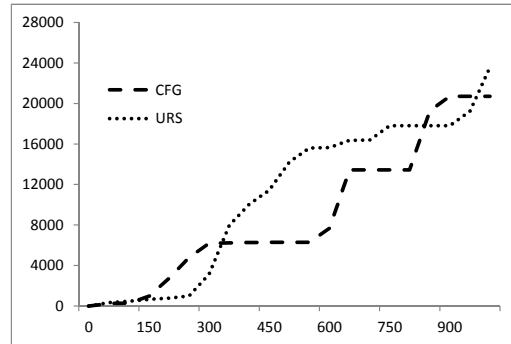
(a)



(b)



(c)



(d)

Figure 4.7: Extra coverage provided for (a) cdaudio (b) diskperf (c) floppy (d) kbfiltr by our method. X-axis: Crest path coverage, Y-axis: Additional path coverage from subsumption.

#### Fourth experiment – Terminating testing

Finally, we discuss a small but important experiment. We wanted to make a “pure” comparison of concolic testing with and without our method, notwithstanding the complications with measuring the number of subsumed paths, random number sequences and repeated paths we encountered in the previous experiments. In other words, we wanted to actively forbid CREST from exploring subsumed trees instead of letting it run and discarding measurements like before. The problem in doing this is that when terminating with a budget, the sequence of paths executed by naive CREST and our method would be different

and hence, incomparable. However, if the testing process terminates having explored the entire search space (i.e., verified the program), the sequence of paths it took to do so does not matter.

We obtained a smaller non-buggy version of `kbfiltr` from the same benchmark suite and ran CREST’s CFG strategy on it with and without our method, this time actively forbidding CREST from exploring subsumed trees. With our aid, CREST was able to completely verify the program in 20 seconds, whereas naive CREST was able to complete only after 256 seconds. This experiment shows concrete evidence that our method indeed accelerates a typical concolic testing strategy such as CFG towards more path coverage, in this case verifying the program much faster than otherwise. Unfortunately, we could not run this experiment on other benchmarks as they contain a prohibitive number of paths and concolic testing could not explore all of them in a reasonable amount of time making it not possible to make such an interesting comparison.

**Remark.** Although we focus on path coverage in this work, it is worthwhile to note that our method can be also used to improve *branch coverage*. In fact, in some of our preliminary experiments we targeted branch coverage and observed that we were achieving the same branch coverage but sometimes faster with our method than without. However, the problem of branch coverage is simpler than path coverage and hence, pruning is not always vital. In these cases, the overhead of interpolation and subsumption may not pay off.

## 4.6 Summary

We attacked the path-explosion problem of concolic testing by pruning redundant paths using interpolation. The main challenge for interpolation in concolic testing is the lack of control of search order. To solve this, we presented the concept of half and full interpolants that makes the use of interpolants sound, and the technique of greedy confirmation that accelerates the formation of full-interpolants thereby increasing the likelihood of subsuming paths. We implemented our method and empirically presented its performance and path coverage gains.

# Chapter 5

## Interpolation-based Verification

The previous chapters explored how symbolic execution can be employed in program analysis (slicing) and testing. Now, we consider the setting of *program verification*, where symbolic execution has recently gained traction [66, 83, 64], primarily due to its ability to avoid infeasible paths. Such infeasible paths cause “spurious” false-positives to be encountered during the verification process, thereby triggering typically expensive refinement mechanisms. By default, symbolic execution will backtrack at once when it encounters an infeasible path (i.e., when unsatisfiability is detected in the path condition) and avoid these false-positives preemptively.

In that regard, symbolic execution by default is *eager*. This eagerness has been considered as a clear advantage of symbolic execution, in comparison with Abstract Interpretation (AI) [27] or Counterexample-Guided Abstraction Refinement (CEGAR) [24], since infeasible paths could block exponentially large symbolic trees in practice. We have also repeatedly seen that the main challenge for symbolic execution is addressing the path explosion problem. The approaches of [66, 83, 59, 64] tackle this fundamental issue by eliminating from the model those facts which are irrelevant or too-specific for proving the unreachability of the error nodes. This “learning” phase consists of computing *interpolants* in the same spirit of no-good learning in SAT solvers.

Informally, in the context of program verification, the interpolant at a given program point can be seen as a formula that succinctly captures the reason of infeasibility of paths which go through that program point. In other words it

succinctly captures the reason why paths through the program point are safe. As a result, if the program point is encountered again through a different path such that its path condition implies the interpolant, the new path can be *subsumed*, because it can be guaranteed to be safe. The previous chapters showed that interpolation is crucial in scaling symbolic execution because it can potentially result in exponential savings by pruning large sub-trees. We have also seen that the efficiency of symbolic execution relies heavily on the “quality” of the computed interpolants. Particularly, the weaker the interpolant is in logical strength, the more powerful it is in subsuming other symbolic states.

This is where a conflict between eagerness and learning arises. Eagerly stopping and backtracking at an infeasible state can make the learned interpolants unnecessarily too *restrictive* – while the interpolant would typically capture the reason for infeasibility of the state, the infeasibility could have nothing to do with the safety of the program. In practice, safety properties often involve a small number of variables whereas conditional expressions, which act as guards by causing infeasibility in paths, could be on any unrelated variable. Ultimately, this causes the (restrictive) interpolant to disallow subsumption in future, mitigating its benefit. In other words, eagerness hinders a *property-directed* approach.

In this chapter, we propose a new method to enhance the learning of powerful interpolants but without losing the intrinsic benefits of symbolic execution. Whenever an infeasible path is encountered, instead of backtracking immediately, we *selectively abstract* (i.e., logically weaken) the infeasible state so that it becomes feasible, and proceed with the search. By performing such an abstraction, we say that we have entered *speculation mode*. More generally, as we progressively abstract away infeasibility from a symbolic path, we are exhibiting a property-directed strategy, i.e., ignoring the infeasibilities along the path until the real reason why the path is safe is found. Note that the sole purpose of speculation is to find better interpolants – we already know any path with an infeasible prefix is safe.

However, since exploration of infeasible states is in general a wasteful effort, we subject the speculation to a *bound*. This mitigates the potential blowup of

the speculative search, while still retaining the possibility of discovering good interpolants. Intuitively, this bound should be linearly related to the program size: anything less than this could make the speculation phase arbitrarily short. It is a main contribution of this work, that in the other direction, a linear bound is *good enough*.

## 5.1 Examples

We first exemplify the case when (eager) symbolic execution is clearly not an efficient way to conduct a proof. For the programs in Fig. 5.1, assume (1) the boolean expressions  $e_i$  do not involve the variables  $x$  and  $y$ , and (2) the desired postcondition is  $y \leq n$  for some constant  $n > 0$ . A *path expression* is of the form  $E_1 \wedge E_2 \wedge \dots \wedge E_n$  where each  $E_i$  is either  $e_i$  or its negation. Note that each of the ( $2^n$ ) path expressions represents a unique path through each of the programs.

<pre>x = y = 0 <b>if</b> (e<sub>1</sub>) y++ <b>else</b> x++ <b>if</b> (e<sub>2</sub>) y++ <b>else</b> x++ ... <b>if</b> (e<sub>n</sub>) y++ <b>else</b> x++</pre>	<pre>x = y = 0 <b>if</b> (e<sub>1</sub>) y += 2 <b>if</b> (e<sub>2</sub>) y += 2 ... <b>if</b> (e<sub>n</sub>) y += 2</pre>
--	---

(a) Lazy is Good

(b) Eager is Good

```
x = y = 0
if (e1) y++ else x++
...
if (ej) y++ else y = n+1
...
if (en) y++ else x++
```

(c) Lazy is Still Better

Figure 5.1: Proving  $y \leq n$ : Eager vs Lazy

Given the first program in Fig. 5.1(a), we can reason that the postcondition  $y \leq n$  always holds, *without considering* the satisfiability of the path expressions. Using symbolic execution, in contrast, many of the unsatisfiable path expressions need to be detected and worse, their individual reasons for unsatisfiability (the “interpolants”) need to be recorded and managed. Note that if we

used a CEGAR approach [24] here, where *abstraction refinements* are performed only when a spurious counter-example is encountered, we would have a very efficient (linear) proof.

In the next program in Fig. 5.1(b), slightly modified from the previous, we present a dual and opposite situation. Note that the program is safe just if, amongst the path expressions that are satisfiable, less than  $n/2$  of these involve a distinct and positive expression  $e_i$  (as opposed to the negation of  $e_i$ ), for  $i$  ranging from 1 to  $n$ . This means that the number of times the “then” bodies of the if-statements are (symbolically) executed is less than  $n/2$ . Here, it is in fact necessary to record and manage the unsatisfiable path expressions as they are encountered during symbolic execution. CEGAR, in contrast, would require a large number of abstraction refinements in order to remove counter-examples arising from not recognising the unsatisfiability of “unsafe” path expressions, i.e. those corresponding to  $n/2$  or more increments of  $y$ .

In practice, a typical program would correspond to being in between the above two extreme cases in Fig. 5.1(a) and 5.1(b). Our key argument, however, is that in fact a typical program lies closer to the first example rather than the second. For the final example program in Fig. 5.1(c), assume that all and only the path expressions which contain the subexpression  $e_j$  are unsatisfiable. (In other words, the only way to execute the  $j^{\text{th}}$  if-statement is through its “then” body.) Here we clearly need to detect the presence of the expression  $e_j$  and not any of the other expressions. More generally, we argue that while some path expressions must be recorded and managed, this number is small. The challenge is, of course, is how to *find* these important path expressions, which is precisely the objective of our speculation algorithm. We next exemplify this.

Consider the program fragment in Fig. 5.2 executed under two different initial contexts:  $y = 4 \wedge x > 0$  and  $y \leq 3 \wedge x \leq 1$ . In both contexts, the program is safe because  $y \leq 100$  at the end. Throughout the example, assume weakest preconditions (WP) are used as interpolants.

Symbolic execution (eager) would start at program point  $\ell_1$  with the first context  $y = 4 \wedge x > 0$ . Assume it first takes the then branch with condition  $x=1$ , executing  $x++$  and reaching  $\ell_4$ . Proceeding along the then branch from

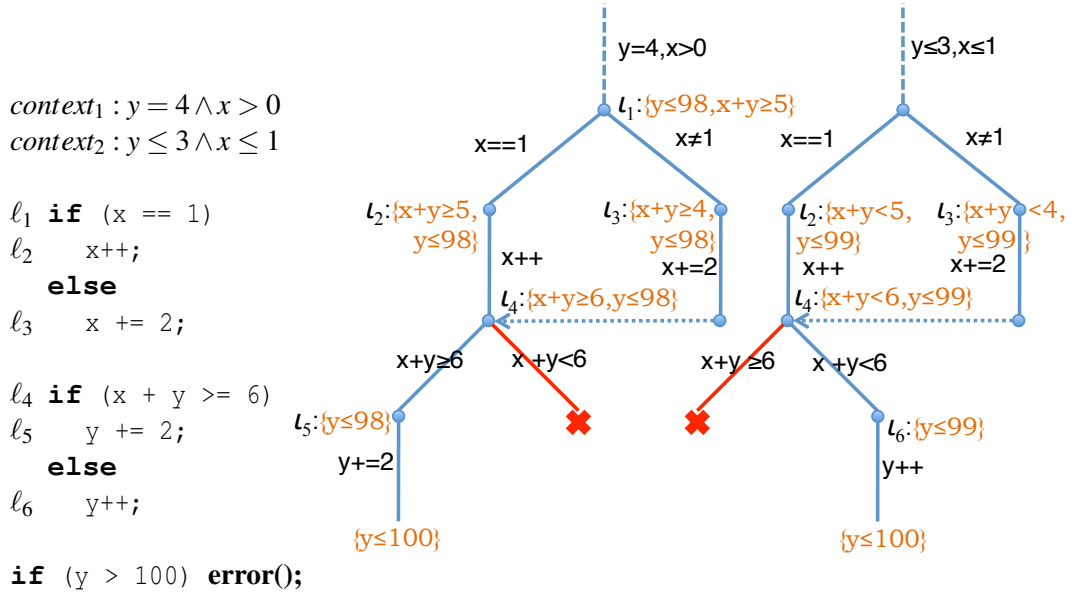


Figure 5.2: A Program and its (Eager) SE Tree with Learning

$l_4$ , it executes  $y+=2$  and reaches the end of the safe path, generating the (WP) interpolant  $y \leq 98$  at  $l_5$ . Now from  $l_4$ , it finds that the else branch is infeasible as the path condition  $y = 4 \wedge x > 0 \wedge x = 1 \wedge x' = x + 1 \wedge x' + y < 6$  is unsatisfiable. Being eager, symbolic execution would immediately backtrack, and to preserve this infeasibility, it would learn the interpolant  $x' + y \geq 6$ . Combining the then and else body's interpolants, it would generate  $x' + y \geq 6 \wedge y \leq 98$  at  $l_4$  (note that in Fig. 5.2 we project the formula on the original variable names). Passing this back through WP propagation would result in  $x + y \geq 5 \wedge y \leq 98$  at  $l_2$ .

Now, executing the else body  $x+=2$  from  $l_1$ , it would reach  $l_4$  with the path condition  $y = 4 \wedge x > 0 \wedge x \neq 1 \wedge x' = x + 2$ , which implies the interpolant  $x' + y \geq 6 \wedge y \leq 98$ . Therefore the path would be *subsumed* (dotted line). Propagating this interpolant through  $x+=2$  would result in  $x + y \geq 4 \wedge y \leq 98$  at  $l_3$ . Now, combining the then and else body's interpolant at  $l_1$  would result in the disjunction:  $(x = 1 \Rightarrow (x + y \geq 5 \wedge y \leq 98)) \wedge (x \neq 1 \Rightarrow (x + y \geq 4 \wedge y \leq 98))$ . For the sake of clarity, we strengthen this to  $y \leq 98 \wedge x + y \geq 5$ , but we assure the reader that our discussion is not affected by this. Thus, the final symbolic execution tree explored for this context will be the one on the left in Fig. 5.2.

Now, when the program fragment is reached along the second context  $y \leq 3 \wedge x \leq 1$ , subsumption cannot take place at  $l_1$  as the context does not imply the

interpolant  $y \leq 98 \wedge x + y \geq 5$ . Symbolic execution would therefore proceed to generate the symbolic tree shown on the right. It is worth noting that even if the program was explored with the order of the contexts swapped, subsumption cannot take place at the top level.

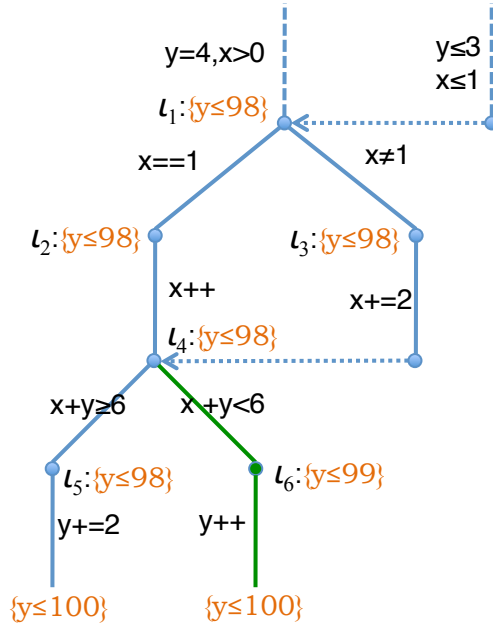


Figure 5.3: Lazy SE Tree with Learning

Consider now our lazy symbolic execution process invoked on this program. We would perform symbolic execution exactly the same as before, except when the unsatisfiable path condition  $y = 4 \wedge x > 0 \wedge x = 1 \wedge x' = x + 1 \wedge x' + y < 6$  is encountered, instead of backtracking, we selectively abstract the formula to make it satisfiable. Since we are doing forward symbolic execution, we selectively abstract by *deleting* the constraint(s) from the latest guard that we encountered (i.e.,  $x' + y < 6$ ) to make the formula satisfiable.<sup>1</sup>

After performing selective abstraction, we enter “speculation mode” with the abstracted path condition  $y = 4 \wedge x > 0 \wedge x = 1 \wedge x' = x + 1$ . A problem now is that in general, the sub-tree underneath the infeasible branch may be exponentially large, exploring which is wasteful as we already know that it is safe. Therefore it is necessary to impose a bound on the speculative search. We remark on our design choice of such a bound in later technical Sections.

Triggering speculation at  $l_4$ , we execute the statement  $y++$  at  $l_6$  and reach the end of the (safe) path. Speculation has now succeeded, hence we annotate  $l_6$  with  $y \leq 99$ . Combining the interpolants at  $l_4$ , we get  $y \leq 98$ . Propagating it back through the tree as shown in Fig. 5.3 we get the interpolant  $y \leq 98$  at  $l_1$ . Now, when the program fragment is reached along the second context  $y \leq$

<sup>1</sup>In principle, selective abstraction can be done in many ways, for instance, by also deleting  $y = 4$ ,  $x' = x + 1$  or any combination. We defer to Section 5.4 the reasoning behind our design choice of deleting the latest guard.



$3 \wedge x \leq 1$ , the interpolant is implied at  $\ell_1$ , and the entire tree can be subsumed at the top level. Note that we applied strengthening of WP as before, but we assure that even without strengthening the subsumption will still take place.

This example has shown that speculation can potentially result in exponential savings. The reason speculation works in practice is that safety properties are only on a small subset of variables whereas program guards that cause infeasibility can be on any of them. Temporarily ignoring the infeasibility helps in discovering interpolants closely related to the safety, such as those in Fig. 5.3, rather than interpolants that blindly preserve the infeasibility, such as those in Fig. 5.2. In Section 5.4, we empirically show that the exponential gains provided by speculation clearly outweigh its cost.

## 5.2 Background

**Symbolic Execution.** For the purposes of this chapter, we make two changes to our formalism of symbolic execution. First, we re-define the symbolic step function  $\text{SYMSTEP}$ , as follows:

$$\mathbf{v}' \equiv \text{SYMSTEP}(\mathbf{v}, t) \triangleq \begin{cases} \langle \ell', s, \Pi \wedge \llbracket c \rrbracket_s \rangle & \text{if } \text{op} \equiv \text{assume}(c) \\ \langle \ell', s[x \mapsto \llbracket e \rrbracket_s], \Pi \rangle & \text{if } \text{op} \equiv x = e \end{cases} \quad (5.1)$$

This definition ensures that  $\text{SYMSTEP}$  will return a symbolic state  $\mathbf{v}'$  regardless of whether it is a feasible state or not. Note that the original definition of  $\text{SYMSTEP}$  in Section 2.1 checked whether  $\Pi \wedge \llbracket c \rrbracket_s$  is satisfiable, and only returned the next state if it was indeed so.

Secondly, we define a “selective abstraction” operator  $\nabla : \text{FOL} \times \text{FOL}$  that accepts an *unsatisfiable* formula  $\Pi$  and returns a satisfiable formula that is an abstraction of  $\Pi$  (i.e., logically weaker than  $\Pi$ ).

The rest of the formalisms, including the definition of interpolants and subsumption check, are as defined in Section 2.2.

**Eager vs. Lazy.** We say that a symbolic execution approach is *eager* if the successor relation  $\text{SUCC}$  is defined only for feasible states. In other words, when

we encounter an infeasible state, we immediately backtrack and compute an interpolant, succinctly capturing the reason of the infeasibility. Though different systems might employ different search strategies for symbolic execution (our formulation above is called *forward* symbolic execution [78]), it is worth to note that all common symbolic execution engines are indeed eager. This eagerness has been considered as a clear advantage of symbolic execution, since it avoids the consideration of infeasible paths, which could be exponential in number.

However, with learning (interpolation), being eager might not give us the best performance. The intuition behind this is as follows. Lemma 1 showed us that if a symbolic state  $v'$  subsumes another state  $v$ , then the tree rooted at  $v$ , say  $T_v$ , has less feasible paths (in a subset sense) than the tree rooted at  $v'$ , say  $T_{v'}$ . Therefore, if  $T_{v'}$  itself has very few feasible paths due to eagerness, it is unlikely that the learned interpolant at  $v'$  would be able to subsume many of such instances of  $T_v$ .

### 5.3 Algorithm

We present our algorithm as a symbolic execution engine with interpolation and speculative abstraction. In Fig. 5.4, the recursive procedure `SymExec` is of the type  $\text{SymExec} : \text{SymStates} \times \mathbb{N} \rightarrow \text{FOL} \cup \{\varepsilon\}$ . It takes two parameters – a symbolic state  $v$  typically on which to do symbolic execution, and a number representing the current level of speculative abstraction, which we will define soon. Its return value is a FOL formula representing the interpolant it generated at  $v$ . A special value of  $\varepsilon$  is used to signify failure of speculation.

Initially, `SYMEXEC` is called with the initial state  $s_0$  with  $\ell_{\text{start}}$  as the program point, an empty symbolic store, and the path condition *true*. For clarity, ignore lines 2-5 which we will come to later. Lines 6-12, represent the three base cases of eager symbolic execution in general – terminal, subsumed and infeasible node (of course, in our lazy method infeasible node is not a base case). In line 6, if the current symbolic state  $v$  is a terminal node (defined by  $\ell$  being the same as  $\ell_{\text{end}}$ ), we simply set the current interpolant  $\bar{\Psi}$  to *true*, as the path is safe and there is no infeasibility to preserve. In line 7, the subsumption check is performed to see

if there exists another symbolic state  $v'$  at the same program point  $\ell$  such that  $v'$  subsumes  $v$  (see Definition 3(a)). If so, the current interpolant  $\bar{\Psi}$  is set to be the same as the subsuming node's interpolant  $\bar{\Psi}'$ . Note that this is an important case for symbolic execution to scale as it can result in exponential savings.

In line 8, we check if the current state  $v$  is infeasible, defined by  $\llbracket v \rrbracket_s$  being unsatisfiable. Normally at this point, eager symbolic execution would simply generate the interpolant *false* to denote the infeasibility of  $v$  and return. For lazy symbolic execution, we begin our speculation procedure here. Line 9 creates a new symbolic state  $v'$  such that it has the same program point  $\ell$  and symbolic store  $s$  as  $v$ , but its (unsatisfiable) path condition  $\Pi$  is selectively abstracted using  $\nabla$  to make the new path condition, which is satisfiable. In our implementation of  $\nabla$ , since SYMEXEC does forward symbolic execution, the path condition

```

Assume initial state  $s_0 \equiv \langle \ell_{\text{start}}, \cdot, \text{true} \rangle$ 
1: Initially : SymExec( $s_0, 0$ )
function SymExec( $v \equiv \langle \ell, s, \Pi \rangle$ , AbsLevel)
2: if AbsLevel > 0 then
3:   if (bounds violated) or ( $\ell \equiv \ell_{\text{error}}$ ) then return  $\varepsilon$  endif
4: else if  $\ell \equiv \ell_{\text{error}}$  then report error and halt
5: endif

6: if TERMINAL( $v$ ) then  $\bar{\Psi} = \text{true}$ 
7: else if  $\exists v' \equiv \langle \ell, \cdot, \cdot \rangle$  annotated with  $\bar{\Psi}$  s.t. SUBSUME( $v, \langle v', \bar{\Psi}' \rangle$ ) then  $\bar{\Psi} = \bar{\Psi}'$ 
8: else if INFEASIBLE( $v$ ) then
9:    $v' = \langle \ell, s, \nabla(\Pi) \rangle$ 
10:   $\bar{\Psi}' = \text{SymExec}(v', \text{AbsLevel} + 1)$ 
11:  if  $\bar{\Psi}' \equiv \varepsilon$  then  $\bar{\Psi} = \text{false}$  else  $\bar{\Psi} = \bar{\Psi}'$  endif
12:  if AbsLevel  $\equiv 0$  then clear data on bounds endif
13: else
14:   $\bar{\Psi} = \text{true}$ 
15:  foreach transition  $t: \ell \rightarrow \ell'$  do
16:     $v' = \text{SYMSTEP}(v, t)$ 
17:     $\bar{\Psi}' = \text{SymExec}(v', \text{AbsLevel})$ 
18:    if  $\bar{\Psi}' \equiv \varepsilon$  then return  $\varepsilon$ 
19:    else  $\bar{\Psi} = \bar{\Psi} \wedge \text{INTP}(\Pi, \text{constraints}(t) \wedge \neg \bar{\Psi}')$ 
20:  endfor
21: endif
22: annotate  $v$  with  $\bar{\Psi}$  and return ( $\bar{\Psi}$ )
end function

```

Figure 5.4: A Framework for Lazy Symbolic Execution with Speculative Abstraction

would have been feasible until the preceding state whose successor is  $v$ . That is, the state  $v''$  such that  $\text{SUCC}(v'', v)$  must have been a feasible state. Hence simply setting  $\Pi$  to the path condition of  $v''$  would make it satisfiable. This mimics *deleting* the latest constraint(s) from  $\Pi$  that caused its infeasibility. In Section 5.4, we discuss the reasons for implementing  $\nabla$  in this way.

Once the abstraction is made, we now speculate by recursively calling SYMEXEC with  $v'$  and incrementing the abstraction level by 1. An abstraction level greater than 0 means that we are under speculation mode. SYMEXEC essentially performs symbolic execution on the selectively abstracted state but with a condition – focus now on lines 2-5. Running under speculation mode, if at any point the bound is violated or if the error location  $\ell_{\text{error}}$  is encountered, it means the speculation failed. In this case, we return a special value  $\varepsilon$  to signify the failure (line 3). Of course, if we are not speculating and  $\ell_{\text{error}}$  is encountered (line 4), then it is a real error to be reported and the entire verification process halts. Otherwise, SYMEXEC proceeds to normally explore  $v$  and finally return an interpolant.

Now in line 10, the interpolant returned from speculation is stored in  $\overline{\Psi}'$ . If  $\varepsilon$  was returned, indicating that speculation failed, we simply resort to using *false* as the interpolant, just like a fully eager symbolic execution procedure. Otherwise, we use the interpolant computed by speculation (line 11). Finally, in line 12, if the current abstraction level is 0 (i.e., we are at the ‘root’ of the speculation tree), then regardless of whether we succeeded or not, we reset all the data that count towards the bounds. For instance, in our implementation, we restrict the speculation to not explore more than one state per program point  $\ell$ , which would result in a bound that is linear in the program’s size. In this case, we have to maintain a count of the number of states explored for each program point. At line 12 this data is cleared since the speculation has finished.

Note that there are two reasons why speculation can fail. A first reason is simply that an abstracted guard is needed to *avoid a counter-example*. If this guard corresponds to abstraction level 0, speculation resulted in nothing learnt *at this program point* (but we could have learnt something from the start of speculation until encountering the counter-example, for descendant program points).

If however the guard abstraction is at a deeper level, the top-level invocation of speculation still can learn new interpolants. The second reason why speculation can fail is that the *bound was exceeded*. In this case, we put forward that, by increasing the bound, it is not likely to result in significant learning. That is, increasing the bound is a strategy of diminishing returns. We will return to this point when we discuss certain statistics in Section 5.4.

If none of the base cases were activated, SYMEXEC proceeds to unwind the path, in lines 13-20. It first initialises the interpolant  $\bar{\Psi}$  to *true*. Then, for every transition from the current program point  $\ell$ , it does the following. First it performs a symbolic step (SYMSTEP) to obtain the next symbolic state  $\nu'$  along the transition  $t : \ell \rightarrow \ell'$ . Then, it recursively calls itself with  $\nu'$  to obtain an interpolant  $\bar{\Psi}'$  for  $\nu'$  (note that we are not speculating here so the abstraction level is unchanged). Now, if the returned interpolant is  $\varepsilon$ , it means further down some speculation resulted in failure. Hence it simply propagates back this failure by returning  $\varepsilon$  (line 18). Otherwise, it computes the current interpolant by invoking INTF on the path condition  $\Pi$  and the conjunction of the constraints of the current transition,  $constraints(t)$ , with the negation of  $\bar{\Psi}'$  (where  $\Pi \wedge constraints(t) \wedge \neg \bar{\Psi}'$  is unsatisfiable). The result is conjoined with any existing interpolant (line 19). Finally, in line 22 the current state  $\nu$  is annotated with the interpolant  $\bar{\Psi}$ , which is then returned. This annotation is persistent such that the subsumption check at line 7 can utilise this information.

**On loop handling:** In the presence of unbounded loops, symbolic execution might not terminate in general. An important contribution of the work [59] is to attempt to make symbolic execution terminate by discovering inductive loop invariants strong enough to prove safety, and unrolling the loop until this can be achieved. We were able to incorporate the loop handling technique in [59] into the implementation of our algorithm without difficulty. Note that while attempting to discover loop invariants to make the symbolic execution finite, [59] can itself go into non-termination, but this issue is orthogonal to our work.

We conclude this section with some insights about the new interpolants discovered by speculation. At the root of speculation, the eager algorithm would have returned *false* as an interpolant. Therefore any other valid interpolant is

clearly better. However, is it the case that using the new (and better) interpolant here, results in better interpolants higher up in the tree? Intuitively the answer is yes, provided that the interpolation algorithm is, in some sense, well behaved. We formalise this as follows.

**Definition 10** (Monotonic Interpolation). *The interpolation method used in our algorithm is said to be monotonic if for all transition  $t$ , path condition  $\Pi$ , and formulas  $\bar{\Psi}_1, \bar{\Psi}_2 \bullet \bar{\Psi}_1 \models \bar{\Psi}_2$  implies  $INTP(\Pi, constraints(t) \wedge \neg \bar{\Psi}_1) \models INTP(\Pi, constraints(t) \wedge \neg \bar{\Psi}_2)$*

Monotonicity ensures that better interpolants at a program point translate into better interpolants at a predecessor point. The supreme interpolation algorithm, based on the weakest precondition, is of course monotonic. A more practical algorithm, however, may not be guaranteed to be monotonic. For example, an algorithm which is based on computing an unsatisfiable core (i.e., simply disregarding constraints which do not affect unsatisfiability), is in general not monotonic because it can arbitrarily choose between choices of cores.

Nevertheless, we noticed in our experiments, detailed in Section 5.4, that new interpolants from speculation do translate into better interpolants and this, in turn, produces more subsumption. This indicates that the interpolation algorithm employed in [64], is indeed relatively well behaved. Some random inspections of the interpolants obtained in the experiments showed that we often have monotonic behaviour in practice, although not theoretically. We show via concrete statistics that as a result of this, we obtain fewer and yet better interpolants.

## 5.4 Experimental Evaluation

We implemented our lazy algorithm on top of TRACER [64], an eager symbolic execution system, and made use of the same interpolation method and theory solver presented in [64]. Let us now remark on our two design choices.

**Selective abstraction:** in principle, selective abstraction ( $\nabla$ ) can be done in many ways, formally, by deleting any “correction subset” [77] of the unsatisfiable formula. We implemented selective abstraction by deleting constraint(s)

from the latest guard that we encountered during forward symbolic execution. The reason for this is two-fold. Firstly, deleting the latest guard *guarantees* the formula to become satisfiable without requiring to compute any of its correction subsets (the latest guard is trivially a *minimal* correction subset), which could be expensive. Secondly, given an *incremental* theory solver, deleting the latest constraint can be implemented more efficiently than deleting one encountered earlier. Nevertheless, although more sophisticated analysis techniques may be employed here to make a well-informed decision, the empirical results show that this approach works well in practice.

**Speculation bound:** we used a *linear* bound for the speculation. In particular, during speculation if a program point is visited more than once and it cannot be subsumed, we stop the speculative search, and use the interpolant *false* at the latest speculation point. Intuitively, anything less than a linear bound could make speculation arbitrarily short, hence we need to give each program point *at least* one chance to be explored. Our experiments confirm that often, a linear bound that gives each program point *at most* one chance, is good enough.

We used as benchmarks sequential C programs from a varied pool – five device drivers from the ntdrivers-simplified category of SV-COMP 2013 [12]: cdaudio, diskperf, floppy, floppy2 and kbfiltr, two Linux drivers qpmouse and tlan, an air traffic collision avoidance system tcas, and two programs from the Mälardalen WCET benchmark [79] statemate and nsichneu for which the safety property was the approximate WCET. All experiments are carried out on an Intel 2.3 Ghz machine with 2GB memory.

To give a perspective of where TRACER stands in the spectrum of verification tools, we compare its performance with two competitive verifiers CPA-CHECKER [106] (ABM version) and IMPACT [80]. Of these, IMPACT implements an interpolation-based model checking procedure, whereas CPA-CHECKER is a hybrid of SMT-based search and CEGAR. Since IMPACT is not publicly available, we use CPA-CHECKER’s implementation of the IMPACT algorithm [80].

For each benchmark, we record in the shaded columns in Table 5.1 the verification time (in seconds) of CPA-CHECKER (CPA), IMPACT (IMP) and TRACER with *eager* symbolic execution (TRACER EAG.), respectively. As it can be seen

Bench- mark	CPA Time (sec)	IMP Time (sec)	TRACER								
			Time (sec)			States			#Interpolants		
			EAG	LZY	Speedup	EAG	LZY	Red.	EAG	LZY	Red.
cdaudio	19	30	41	23	1.78	4396	2864	35%	3854	2689	30%
diskperf	28	149	53	19	2.79	4309	1617	62%	4012	1514	62%
floppy	27	36	25	12	2.08	3535	1635	54%	3208	1534	52%
floppy2	98	40	42	29	1.45	5063	3153	38%	4536	2863	37%
kbfiltr	3	8	4	3	1.33	973	756	22%	860	691	20%
qpmouse	3	8	32	15	2.13	1313	779	41%	1199	723	40%
tlan	T/O	T/O	41	26	1.58	3895	2545	35%	3542	2324	34%
nsichneu	5	41	40	5	8.00	4481	1027	77%	4379	1018	77%
statemate	2	T/O	72	5	14.40	6680	616	91%	4370	471	89%
tcas	2	11	19	1	19.00	5500	369	93%	5248	348	93%
<b>Total</b>	<b>367</b>	<b>683</b>	<b>369</b>	<b>138</b>	<b>2.67</b>	<b>40145</b>	<b>15361</b>	<b>62%</b>	<b>35208</b>	<b>14175</b>	<b>60%</b>

Table 5.1: Verification Statistics for Eager and Lazy SE (A T/O is 180s (3 mins))

TRACER is generally competitive with IMPACT (faster in overall timing) but sometimes slower than CPA-CHECKER so it can be roughly positioned between the two (closer to CPA-CHECKER) in terms of performance. This comparison is to show that we chose a competitive verifier to implement our algorithm and we fully expect the same benefits to be provided to other similar verifiers.

We now present the main results in the rest of Table 5.1. In the set of columns labelled Time (sec) we show the verification time of TRACER in seconds for each benchmark. In this, the (shaded) column EAG which we just saw, performed eager symbolic execution, while the LZY column performed lazy symbolic execution, and Speedup is the ratio of the two. It can be seen that in all programs, laziness makes the verification much faster, providing an average speedup of 2.67. This also makes lazy TRACER perform much better than eager TRACER. We notice enormous improvement for nsichneu, statemate and tcas, as these are programs with a large number of infeasible paths and the safety property on a small number of variables, the perfect scenario for our speculation to shine.

We move on to a more fine-grained measurement than time in the next set of columns States, which shows the number of symbolic states TRACER encountered during verification. In total, we found that 40145 states were encountered without speculation (EAG) and just 15361 states with speculation (LZY), a reduction of about 62%. This shows that speculation is resulting in more sub-



sumption, which thereby causes a reduction in the search space.

Next, we measure the improvement in memory provided by speculation. In the set of columns `#Interpolants`, we show the total number of interpolants stored by TRACER at the end of the verification process. Interpolants typically contribute to a major part of memory used by modern symbolic execution verifiers. In this regard, *laziness* reduced the number of interpolants in TRACER from 35208 (EAG) to 14175 (LZY), a reduction of 60% across all benchmarks.

We focus on the two metrics seen above: number of interpolants (`#Interpolants`), and amount of subsumption, in terms of states (`States`) encountered. The critical point is the inverse relationship: *laziness provided a much smaller number of interpolants while simultaneously increasing subsumption*. In other words, the *quality* of interpolants discovered through speculation is enhanced.

We conclude this section with a few more statistics which, while not directly linked to absolute performance, nevertheless shed additional insight. First, consider the number of distinct program variables that are involved in the interpolants. In the case without speculation, we noticed across all benchmarks that there were **363** such variables. In contrast, with speculation, the number is only **229**. This means that many (134) variables were not required to determine the safety of the program. They were being needlessly tracked by interpolants simply to preserve infeasible paths.

Next consider the “success rate” of speculation: how often does speculation find an alternative interpolant? For simplicity, consider only those speculations triggered at the top-level of the algorithm (from abstraction level 0 to 1). We found, across the benchmark programs, a rate of **40-90%**, more often at the higher end. This means that speculation returns something useful most of the time. However, note importantly that even when speculation was not successful at the top-level, there is likely to have been interpolants discovered at the lower levels. These are interpolants one would have not found without speculation.

To elaborate on the success rate of 40-90%, programs having large number of infeasible paths tend to produce a high success rate, because as per our key intuition, many such paths will be unrelated to the safety. Similarly, programs with few infeasible paths produce a low success rate. In our experiments, the

highest success rates (90%) were from `nsichneu`, `tcas` and `statemate`, which have a large number of infeasible paths as mentioned before.

Finally, reconsider the bound. The above success rate also indicates that there are a significant, though minor, number of failures. We wish to mention that when we do fail, the overwhelming reason is *not the bound*, but instead, the (spurious) counterexamples. In summary, the rather high rate of success, and the rather low rate of failure caused by the bound, together suggest that increasing the bound would be a strategy of diminishing returns.

## 5.5 Related Work and Discussion

Symbolic execution [71] has been widely used for program understanding and program testing. We name a few notable systems: KLEE [17], Otter [88], and SAGE [50]. Traditionally, execution begins at the first program point and then proceeds according to the program flow. Thus symbolic execution is actually *forward* execution. Recently, [78] proposed a variation, *directed* symbolic execution, making use of heuristics to guide symbolic execution toward a particular target. This has shown some initial benefits in program testing.

For the purpose of having scalability in program verification, however, symbolic execution needs to be equipped with *learning*, particularly in the form of interpolation [66, 83, 59, 64, 6]. Due to the requirement of *exhaustive* search, as in the case of this work, these systems naturally implement forward symbolic execution. All the above-mentioned systems can be classified as *eager* symbolic execution. In other words, we do not continue a path when the accumulated constraints are enough to decide its infeasibility.

In the domain of SAT solving and hardware verification, *property directed reachability* (PDR) [38] has recently emerged as an alternative to interpolation [81]. Some notable extensions of PDR are [56, 22, 105]. However, the impact of PDR to the area of software verification is still unclear. While such “backward” execution has merits in terms of being goal directed, it has lost the advantage of using the (forward) computation to limit the scope of consideration.

In contrast, our lazy symbolic execution preserves the intrinsic benefits of

symbolic execution while at the same time, by opening the infeasible paths selectively, it enables the learning of *property directed* interpolants. We believe this is indeed the reason for the efficiency achieved and demonstrated in Section 5.4.

The traditional CEGAR-based approach to verification may also be thought of “lazy”. This is because it starts from a coarsely abstracted model and subsequently refines it. Such concept of laziness is, therefore, different from what discussed in this work. In the context of this work, given a refined abstract domain, a CEGAR-based approach is in fact considered as eager, since it avoids traversal of infeasible paths, which are blocked by the abstract domain. Some of such paths are indeed counter-examples learned from the previous phases. The work [83] discussed this as a disadvantage of CEGAR-based approaches: they might not recover from over-specific refinements. Our contribution, therefore, is plausibly applicable in a CEGAR-based setting.

There is now an emerging trend of employing generic SMT solvers for (bounded) symbolic execution, and since modern SMT solvers, e.g. [34], do possess the similar power of interpolation – in the form of conflict clause learning or lemma generation – we now make a few final comments in this regard.

First, note that lazy symbolic execution has no relation with the concept of *lazy* SMT. In particular, the dominating architecture  $DPLL(T)$ , which underlies most state-of-the-art SMT tools, is based on the integration of a SAT solver and one (or more)  $T$ -solver(s), respectively handling the Boolean and the theory-specific components of reasoning. On the one hand, the SAT solver enumerates truth assignments which satisfy the Boolean abstraction of the input formula. On the other hand, the  $T$ -solver checks the consistency in  $T$  of the set of literals corresponding to the assignments enumerated. This approach is called lazy (encoding), and in contrast to the eager approach, it encodes an SMT formula into an equivalently-satisfiable Boolean formula and feeds the result to a SAT solver. See [96] for a survey.

Second, we note that though the search strategies used in modern  $DPLL$ -based SMT solvers would be more dynamic and different from the forward symbolic execution presented in this work, it is safe to classify these SMT solvers

as *eager* symbolic execution. This is because, in general, whenever a conflict is encountered, a DPLL-based algorithm would analyse the conflict, learn and/or propagate new conflict clauses or lemmas, and then immediately backtrack (backjump) to some previous decision, dictated by its heuristics [32].

We believe that for the purpose of program verification, the benefit of being lazy by employing speculative abstraction, would also be applicable to SMT approaches. This is because, in general, we can always miss out useful (good) interpolants if we have not yet seen the complete path. In this work, we have demonstrated that in verification, property directed learning usually outperforms learning from “random” infeasible paths. Eagerly stopping when the set of constraints is unsatisfiable might prevent a solver from learning the conflict clauses which are more relevant to the safety of the program. In SMT solvers, the search, however, is structured around the decision graph. Therefore, some technical adaptations to our linear bound need to be reconsidered. For example, a bound based on the number of decisions seems to be a good possibility.

## 5.6 Summary

We presented a systematic approach to perform speculative abstraction in symbolic execution in pursuit of program verification. The basic idea is simple: when a symbolic path is first found to be infeasible, we abstract the cause of infeasibility and enter speculation mode. In continuing along the path, more abstractions may be performed, while remaining in speculation mode. Crucially, speculation is only permitted up to a given bound, which is a linear function of the program size. A number of reasonably sized and varied benchmark programs then showed that our speculative abstraction produced speedups of a factor of two and more.

# Chapter 6

## Trace Understanding

Programmers often have to reason about execution traces during the software development process. Especially in debugging, when an input violates an assertion and the corresponding error trace is obtained, programmers have to understand why the input failed. This is a laborious process since execution traces are typically very long due to loops the input has iterated through. Traditionally, dynamic slicing [72] was used to remove irrelevant statements that do not contribute to the assertion, using dependency information. More recently techniques such as [39, 20] have been proposed to use more intelligent semantic information to remove irrelevant statements. However, none of the above methods directly address the practical problem of long loop iterations in the trace.

We propose a novel method of compressing loop iterations in the trace by discovering loop invariants. Consider as a motivating example the program below:

```
i=j=0
while (i != n) do
    i=i+1
    j=j+2
end
TARGET: {j ≥ n}
```

Running this program with input  $n=100$  would produce a trace that assigns  $i$  and  $j$  to 0, iterates through the loop 100 times, in the end assigning  $i$  and  $j$  to 100 and 200 respectively, thus implying the **TARGET**<sup>1</sup> that  $j \geq n$ . Now, to understand why

---

<sup>1</sup>In general the target can be any formula that is implied by the trace. For an error trace, the target would be the negation of the violated assertion.

the trace implies the target the programmer needs to go through the 100 iterations. Note that dynamic slicing does not help as the statement  $j=j+2$  is control-dependent on  $i$  through the loop condition, which in turn is data-dependent on  $i=i+1$ , and so no statement would be sliced away.

Our method would produce the following compression and explanation of the trace:

$i=j=0$	<b>INV:</b> $\{j \geq i\}$
<b>INV:</b> $\{j \geq i\}$	—————→
$i == n$	$i=i+1$
<b>TARGET:</b> $\{j \geq n\}$	$j=j+2$
	<b>INV:</b> $\{j \geq i\}$

On the left is the actual compressed trace. It explains that initially the trace assigned  $i$  and  $j$  to 0. Then, on reaching the loop the *invariant* that  $j \geq i$  held at the loop header at every iteration, and eventually the exit condition  $i==n$  was taken. These two pieces of information entail (logically) that  $j \geq n$ , the target. On the right, we provide an explanation for the fact that  $j \geq i$  was indeed a loop invariant, in the form of a Hoare-triple [55]. Starting with the “pre-condition”  $j \geq i$ , executing the statements in the loop body results in the “post-condition”  $j \geq i$  again, thus the formula is invariant. The compressed trace and explanation are much shorter than the original trace and easier to understand as they capture very closely the intuition a programmer would have.

On the surface, this technique may sound quite simple. However, a number of challenges arise when delving deep into it:

### A. What kind of invariants do we need to discover?

In the above example, all the following formulas are invariants for the loop:  $i \geq 0$ ,  $0 \leq j \leq 2n$ , *true* (trivially an invariant for any loop), etc. But not all invariants can be used for this purpose – the invariant used in the compressed trace needs to *imply the target* in the end, similar to the original trace. Only if it does so, in which case we call it a “safe invariant”, we can claim the compressed trace is an explanation of the original trace. For instance, the invariant  $0 \leq j \leq 2n$  is not safe because combined with the loop’s exit condition  $i==n$ , it does not imply the target  $j \geq n$ . Thus, discovering safe invariants is not trivial

and is a primary challenge. This is made even more complicated in the presence of nested loops in the trace, when we may encounter an inner loop while discovering an invariant for the outer loop.

### **B. What if we are unable to discover a safe invariant?**

In the above contrived example, the invariant  $j \geq i$  conveniently happened to be safe. But in many cases, a safe invariant could be very hard to discover. In our experiments in Section 6.4, we noticed several state-of-the-art invariant generators using powerful abstract domains unable to find a safe invariant for our benchmarks. Thus, we need measures to handle such cases when the discovered invariant turns out to be unsafe. In our method, we dynamically *unroll* the loop and attempt the invariant discovery at the next iteration. We found that this is a very reasonable measure in practice as many loops converge towards an invariant as they iterate. For instance, loops typically either keep increasing or decreasing values of variables, and after a few unrolls we can discover lower or upper bounds on those variables, forming an invariant. Again, nested loops make the unrolling more complicated, as we will see in Section 6.

### **C. What if a discovered safe invariant is not so relevant to the target?**

When discovering invariants, we have the choice to aim for stronger or weaker invariants (logically), both of which have pros and cons. Weaker invariants are less likely to be safe because they abstract (hide) more information about the loop. Stronger invariants may carry too much information about the loop, many of which may be irrelevant to the target, making the loop explanation harder to understand. For instance, in the above example,  $j \geq i \wedge j \leq 2n$  is a stronger invariant than just  $j \geq i$  but in fact,  $j \leq 2n$  is not needed to imply the target.

Our method provides a fine balance between the two – first, we aim to discover strongest invariants in order to increase the likelihood of being safe, and once we reach the target along the trace, we *generalise* (weaken) the discovered invariants, removing information from them that were not really needed to imply the target. Thus our method provides the benefits of both worlds.

#### D. Ultimately, is the compressed trace with invariants easier to understand than the original trace?

The “explanation” is provided as Hoare-triples of the form  $\{P\}S\{Q\}$  where executing  $S$  with the pre-condition  $P$  results in the post-condition  $Q$ . If  $S$  is a basic block of code as in the above example, this can be proof checked easily. More complex triples are obtained by combining such triples using the standard Hoare inference rules [55]. The complexity of the explanation therefore depends on the number of triples produced. In the worst case however, the number of triples for the proof can be exponential in the size of  $S$  if it is an arbitrary piece of code. Thus, we need measures to keep the number of triples from growing exponentially.

Consider a more sophisticated example, which we will use to elaborate our method. This example is taken from the benchmark program `for_bounded_loop1.c` from the Software Verification Competition 2013 (SV-COMP13) [12].

```
l1 x=0, y=0, err=0 /* assume n > 0 */
l2 for (i=0; i < n; i++) do
l3     x = x-y
l4     if (x != 0) err = 1 else skip
l5     y = read() /* assume read does not return a 0 */
l6     x = x+y
l7     if (x == 0) err = 1 else skip
      end
l8 TARGET: {x ≠ 0, err ≠ 1}
```

Assume that the program is executed with input  $n=10$  and  $y$ , read at  $l_5$ , is never input as 0. This will produce a trace that iterates through the loop 10 times, and exits with  $x$  being non-zero and  $err$  being 0, implying the target  $\{x \neq 0, err \neq 1\}$ .

Our method performs *symbolic execution* on the trace, collecting the constraints into a *symbolic state* at each location. For instance, the symbolic state at  $l_2$  would be the set of constraints  $\{x = 0, y = 0, err = 0, i = 0\}$ . When a loop is encountered, it attempts to discover an invariant. While technically any invariant generation algorithm can be used in this step, we describe a specific one suitable to our overall method. Briefly, we symbolically execute every path in



the loop exercised by the trace, and delete individual constraints from the loop header’s symbolic state that do not still hold at the end of each path. The whole process is repeated till no deletions are made, i.e., a fixpoint is reached at the loop header, at which point the remaining constraints form the loop invariant. We call this technique *basic individually invariant discovery* (BIID).

BIID works well with our unrolling mechanism because the symbolic state at the loop header is very likely to change after unrolling, at which point BIID keeps individual constraints that are invariant *considering the changes in the new state*. This is very important as unrolling may expose constraints that were not originally invariant before but are indeed invariant from now on. Hence it is a “basic” requirement to at least check if individual constraints in the new symbolic state are now invariant. More sophisticated methods may consider *disjunctions* of constraints, linear relationships between variables, user-provided predicates, etc., which may result in better (stronger) invariants. Such sophistications are indeed admitted in our algorithm, but we found that in practice BIID is fast and yet smart enough to capture safe invariants for our benchmarks. Note that BIID is dependent on the internal representation of a symbolic state in choosing the “candidate” invariant constraints. For instance, the symbolic states  $\{v = 5, w = v\}$  and  $\{v = 5, w = 5\}$  are semantically equivalent, but BIID may return different invariants for each state (if, for example,  $\{w = v\}$  is not invariant in the loop but  $\{w = 5\}$  is). This of course depends on the implementation of the underlying symbolic execution engine.

In our example, let us see how BIID finds an invariant at  $\ell_2$ . Since both  $x$  and  $y$  are zero at  $\ell_2$ ,  $x=x-y$  makes no change to the state, so we reach  $\ell_4$  with the state  $\{x = 0, y = 0, err = 0, i = 0\}$ . This makes the branch condition *false*, so we continue along the path to read  $y$ , which we assume is non-zero. Since  $y$  is now non-zero and  $x$  is zero, executing  $x=x+y$  at  $\ell_6$  results in the state  $\{x = y, y \neq 0, err = 0, i = 0\}$  at  $\ell_7$ . This makes the branch condition *false*, and we reach the looping point  $\ell_2$  again with the state  $\{x = y, y \neq 0, err = 0, i = 1\}$ .

Recall that the initial symbolic state at  $\ell_2$  was  $\{x = 0, y = 0, err = 0, i = 0\}$ . From this, BIID deletes constraints that do not hold anymore.  $x = 0$  and  $y = 0$  do not hold because both  $x$  and  $y$  have now become non-zero, so they are deleted.

$\ell_2$	$\{x = y, y \neq 0, err = 0\}$	$i < n$	$\{x = y, y \neq 0, err = 0\}$	$\ell_3$
$\ell_3$	$\{x = y, y \neq 0, err = 0\}$	$x = x - y$	$\{x = 0, y \neq 0, err = 0\}$	$\ell_4$
$\ell_4$	$\{x = 0, y \neq 0, err = 0\}$	<b>if</b> $(x \neq 0)$ $err = 1$	$\{x = 0, y \neq 0, err = 0\}$	$\ell_5$
$\ell_5$	$\{x = 0, y \neq 0, err = 0\}$	$y = \mathbf{read}()$ and $\mathbf{read}()$ returns non-zero	$\{x = 0, y \neq 0, err = 0\}$	$\ell_6$
$\ell_6$	$\{x = 0, y \neq 0, err = 0\}$	$x = x + y$	$\{x = y, y \neq 0, err = 0\}$	$\ell_7$
$\ell_7$	$\{x = y, y \neq 0, err = 0\}$	<b>if</b> $(x == 0)$ $err = 1$	$\{x = y, y \neq 0, err = 0\}$	$\ell_2$

Figure 6.1: Hoare triples generated for the program `for_bounded_loop1.c`

Similarly,  $i = 0$  does not hold because it has been incremented to 1.  $err = 0$  still holds, and so we get the new state  $\{err = 0\}$  at  $\ell_2$ . Since we did not reach fixpoint (i.e., some deletions were made), we repeat this process again with the new state, but it can be clearly seen that without constraints on  $x$  and  $y$ , the branches at  $\ell_4$  and  $\ell_7$  can also be evaluated to *true*, and so  $err = 0$  also cannot be invariant. Therefore we eventually delete it to get the invariant *true*, which fails to imply the target  $\{x \neq 0, err \neq 1\}$  (i.e., it is unsafe).

This triggers our failsafe that basically discards all the above work, *unrolls* the loop and attempts the same process at the next iteration. It is as though we roll back in time and do not try to discover an invariant, because we now know that it will fail. In other words, we follow the trace till the end of the first iteration, executing the statements from  $\ell_2$  to  $\ell_7$  and then reaching  $\ell_2$  this time with the state  $\{x = y, y \neq 0, err = 0, i = 1\}$ . Now, we trigger BIID again with this state. Since  $x$  and  $y$  are equal,  $x = x - y$  at  $\ell_3$  results in the state  $\{x = 0, y \neq 0, err = 0, i = 1\}$  at  $\ell_4$ . This makes the branch condition at  $\ell_4$  false, so we read  $y$  (non-zero) at  $\ell_5$ , which makes no change to the state. Therefore the statement  $x = x + y$  produces the state  $\{x = y, y \neq 0, err = 0, i = 1\}$  at  $\ell_7$ , which makes the branch condition false. Thus we reach the loop header  $\ell_2$  with the state  $\{x = y, y \neq 0, err = 0, i = 2\}$  after incrementing  $i$ .

Comparing this with the original state  $\{x = y, y \neq 0, err = 0, i = 1\}$  at  $\ell_2$ , BIID would only delete the constraint  $i = 1$  to end up with the invariant  $\{x = y, y \neq 0, err = 0\}$ . This is a safe invariant because combined with the rest of the trace after the loop (i.e., the exit condition  $i \geq n$ ), it implies the target  $\{x \neq 0, err \neq 1\}$ . Thus our method compresses the remaining 9 iterations of the loop. The final compressed trace returned is shown below:

```

 $\ell_1$   $x=0, y=0, err=0$ 
 $\ell_2$   $i=0, i < n$ 
 $\ell_3$   $x = x-y$ 
 $\ell_4$   $x == 0, \mathbf{skip}$ 
 $\ell_5$   $y = \mathbf{read}(), y \neq 0$ 
 $\ell_6$   $x = x+y$ 
 $\ell_7$   $x \neq 0, \mathbf{skip}$ 
 $\ell_2$  INV:  $\{x = y, y \neq 0, err = 0\}$ 
 $\ell_2$   $i \geq n$ 
 $\ell_8$  TARGET:  $\{x \neq 0, err \neq 1\}$ 

```

To provide an explanation that the formula **INV** is indeed an invariant, we will generate the Hoare triples shown in Fig. 6.1.

Note that the invariant (underlined) holds at  $\ell_2$  at the beginning and at the end after going through the loop. We believe the compressed trace together with the triples is a better explanation of the trace than the original trace itself. Instead of going through 10 iterations, the programmer needs to only go through 1 iteration and understand using the Hoare triples the safe invariant that implies the target.

It is important to note that for the above example, all constraints in the discovered invariant are needed to imply the target. One may be tempted to replace  $\{x = y, y \neq 0\}$  with  $\{x \neq 0\}$ , as the target only talks about  $x$ , not  $y$ . However  $\{x \neq 0\}$  is **not** an invariant for the loop. That is, starting with the pre-condition  $\{x \neq 0\}$ , one cannot generate valid Hoare-triples through the loop body such that  $\{x \neq 0\}$  still holds at the end. Thus, the current invariant cannot be generalised any further<sup>2</sup>. However an invariant generated by a different method than ours could contain, for instance, the invariant  $\{i \leq n\}$ , which can be removed since it is not needed to imply the target.

## Compressing Traces with Nested Loops

Nested loops pose important technical challenges to our method because when attempting invariant discovery for the outer loop, we may encounter another inner loop. Should we recursively attempt invariant discovery for the inner loop?

<sup>2</sup>In general, there may be other forms of weakening such as *slackening* [66]  $err = 0$  to  $err \leq 0$ , which is still a safe invariant. Such general forms of weakening are also admitted in our algorithm.

<pre> l1 i=0 l2 while (i &lt; 10) do l3     x=0, j=i l4     while (j &lt; 10) do l5         if (j ≥ 1) x=1 else x= -1 l6         j=j+1 l7     end l7     i=i+1 l8 end l8 for (y=0, k=0; k &lt; 10; k++) do l9     y=y+x l9 end l10 TARGET: {y ≥ 0} </pre>	<pre> l1 i=0 l2 i &lt; 10 l3     x=0, j=i l4     INV: {i = 0} l4     j ≥ 10 l7     i=i+1 l2 i &lt; 10 l3     x=0, j=i l4     INV: {i = 1, j ≥ i, x ≥ 0} l4     j ≥ 10 l7     i=i+1 l2 INV: {i ≥ 1, x ≥ 0} l2 i ≥ 10 l8 y=0, k=0 l8 INV: {i ≥ 1, x ≥ 0, y ≥ 0, k ≥ 0} l8 k ≥ 10 l10 TARGET: {y ≥ 0} </pre>
(a)	(b)

Figure 6.2: (a) Program with nested loops (b) Its compressed trace

Should we unroll the inner loop if we are unable to discover a safe invariant? More importantly, what is the meaning of a safe invariant for the inner loop, and its relation with the outer loop’s invariant? We exemplify these issues with the program in Fig. 6.2(a).

There are two nested loops, the outer one running  $i$  from 0 to 10 and the inner one running  $j$  from  $i$  to 10 every iteration. The inner loop assigns a value to  $x$ , namely 1 or -1, depending on the value of  $j$  being greater than or equal to 1. There is a final loop outside the nest that adds  $x$  to a variable  $y$  (initialised to 0) 10 times, and the target is that  $y$  is non-negative. This of course mandates that the value of  $x$  is non-negative when this loop is executed. Note that for simplicity, we have grounded the inputs into the program. When the program is executed, it would produce a trace containing 10 outer loop iterations, and for each,  $10-i$  inner loop iterations (i.e., 55 iterations), followed by the third loop, for a total of 65 iterations.

Suppose that our algorithm is invoked on this program<sup>3</sup>. It would follow the trace till  $\ell_2$  when it encounters the loop with the state  $\{i = 0\}$ . We say we are at “trace level” whenever we are following the trace, executing statements

<sup>3</sup>For demonstration purposes, we assume that we are allowed to slacken [66] constraints in this example. That is, we consider a constraint  $\{v = w\}$  as the conjunction  $\{v \leq w, v \geq w\}$  so that BIID has more candidates to test for invariance.

grounded by the input. At  $\ell_2$  we would invoke BIID for the loop by symbolically executing all paths in its body. At this point, we depart from trace level as we are not guided by the input anymore. During this BIID attempt, we would encounter the inner loop at  $\ell_4$  with the state  $\{i = 0, x = 0, j = i\}$ . Now, since we are not at trace level, it is *required* for us to find an invariant for the inner loop in order to proceed symbolically executing the rest of the outer loop's body. That is, *we cannot unroll the inner loop during the process of discovering an invariant for the outer loop*, because in general, unrolling when not at trace level (i.e., bounded by the input) can lead to unbounded unrolling.

The good news is that this (recursive) BIID procedure has the liberty to produce *any* valid invariant for the inner loop. Our aim here is not to discover a *safe* invariant, but to simply produce an invariant so that we may discover an invariant for the outer loop that is at trace level. In the worst case, this procedure can return *true*, but it is good to try to discover strongest invariants as much as possible because the inner loop's invariant indirectly affects the strength of the outer loop's invariant. In this example, BIID would discover the inner loop invariant  $\{i = 0, j \geq i\}$  given the state at  $\ell_4$  (we slackened  $j = i$  to  $\{j \leq i, j \geq i\}$  of which the latter was invariant), but we cannot say anything about  $x$  as it may be greater than or less than 0. Thus the state at  $\ell_7$  is  $\{i = 0, j \geq i\}$ , and after executing  $i = i + 1$ , we reach the loop header  $\ell_2$  with the state  $\{i = 1\}$ . We dropped the constraint  $\{j \geq i\}$  as  $i$  has been incremented (more sophisticated methods may still track constraints such as  $\{j \geq i - 1\}$ , but this does not change our discussion of this example).

Comparing this with the original state  $\{i = 0\}$  at  $\ell_2$ , BIID would produce  $\{i \geq 0\}$  as the outer loop invariant. Since we are back at trace level, we now check if this outer loop invariant is safe. To check if an invariant is safe, we symbolically execute the rest of the trace after the loop starting with the invariant as the symbolic state. If the target is implied at the end, the invariant is safe, otherwise it is not. Here, since we discovered an invariant for the outer loop at  $\ell_2$ , the rest of the trace would contain 10 iterations of the loop at  $\ell_8$ . Executing it starting with the state  $\{i \geq 0\}$  would certainly not imply  $\{y \geq 0\}$  at the end, because the invariant failed to capture any constraint on  $x$ , which is added to  $y$ .

Our failsafe mechanism now activates, and begins to unroll the outer loop's first iteration. The inner loop is now lifted to trace level, as though the outer loop's first iteration was placed outside the loop in the program. Now, we would again reach  $\ell_4$  with the state  $\{i = 0, x = 0, j = i\}$ , and discover the invariant  $\{i = 0\}$ . But this time, since we are at trace level, we have to check if this is safe. Note that we now discovered an invariant only for the inner loop at  $\ell_4$ , and so the rest of the trace would still contain 9 outer loop iterations. This would finally assign  $x$  to 1 and cause  $y$  to be incremented, implying the target. Hence,  $\{i = 0\}$  is indeed a safe invariant for the inner loop at trace level. Thus, even though we were unable to compress the outer loop immediately, we unrolled its first iteration and managed to compress the inner loop in that iteration with the invariant  $\{i = 0\}$ .

Next, we follow the trace executing  $i=i+1$ , and reach the second iteration of the outer loop at  $\ell_2$  with the state  $\{i = 1\}$ . Note what happens this time, when we again trigger BIID at  $\ell_2$ . We reach  $\ell_4$  with the state  $\{i = 1, x = 0, j = i\}$ . As before,  $j$  is indeed incremented in the loop, but the symbolic state guarantees that it will always be greater than or equal to 1. This makes the "else" branch at  $\ell_5$  an *infeasible* path, which ensures that  $x$  would never be assigned -1. Therefore, this time we discover the invariant  $\{i = 1, j \geq i, x \geq 0\}$  for the inner loop (we could not have deduced this the first time because the state at  $\ell_4$  was  $\{i = 0, x = 0, j = i\}$ , which does not make the "else" branch infeasible). Now, executing  $i=i+1$ , we reach  $\ell_2$  with the state  $\{i = 2, x \geq 0\}$ . Now, we can discover the outer loop invariant  $\{i \geq 1, x \geq 0\}$  for its remaining iterations. This is a *safe* invariant, as executing the rest of the trace (i.e., the loop at  $\ell_8$ ) with this state will imply the target  $\{y \geq 0\}$ .

Thus, the compressed trace, as shown in Fig. 6.2(b) would contain 2 iterations of the outer loop, both of which contain a compression of the inner loop with the invariants  $\{i = 0\}$  and  $\{i = 1, j \geq i, x \geq 0\}$  respectively, followed by a compression of the outer loop with the invariant  $\{i \geq 1, x \geq 0\}$ . It can be seen that symbolically executing the third loop at  $\ell_8$  with this state, BIID can discover the (safe) invariant  $\{i \geq 1, x \geq 0, y \geq 0, k \geq 0\}$ , and compress it without any unrolling.

Now, our invariant generalisation proceeds as follows. It computes the *weakest precondition* (WP) along the compressed trace starting from the end. At loop headers, it attempts to weaken the discovered invariant as long as the postcondition is still implied. For instance, the weakest precondition of the trace in Fig. 6.2(b) at  $\ell_8$  is  $\{y \geq 0\}$ . Now, it deletes constraints from the invariant  $\{i \geq 1, x \geq 0, y \geq 0, k \geq 0\}$  as long as the resulting formula is still invariant and implies  $\{y \geq 0\}$ . This way, it can delete  $\{i \geq 1\}$  and  $\{k \geq 0\}$  as the resulting invariant  $\{x \geq 0, y \geq 0\}$  implies  $\{y \geq 0\}$ . However it cannot delete  $\{x \geq 0\}$  because without it,  $\{y \geq 0\}$  cannot be invariant, and hence cannot imply the postcondition. Thus, it ends up with the now generalised invariant  $\{x \geq 0, y \geq 0\}$  at  $\ell_8$ . The WP of this formula is then passed up along the trace, and the process repeats till the top is reached. Note that for this trace, no more invariants can be generalised using WP.

The proof for invariance of the **INV** formulas is in the form of a *symbolic execution tree*, which is a tree of all paths explored during symbolic execution of the loop’s body. In Section 6.3, we describe a fairly straightforward process of extracting Hoare triples from symbolic execution trees, using this trace as an example. There we will also see that many Hoare triples are in fact much easier to understand than others because of what is called the “frame-rule”.

## 6.1 Related Work

The works that try to explain traces by removing irrelevant information are the most closely related ones to ours. Dynamic slicing [72, 111, 110] is the traditionally used technique that uses dependency information to remove statements from the trace not contributing to the target. Some enhancements to the pruning power of dynamic slicing were proposed in [109], using “confidence” metrics to rank statements based on their likelihood of being responsible for the error. However, compared to loop invariants, dependency information is limited in its ability to compress loop iterations. In our example above, dynamic slicing cannot remove any iteration, as the target variable  $x$  is data-dependent on  $y$  and (thereby) control-dependent on  $i$  in each iteration.

Recently, more intelligent methods to find irrelevant statements in the trace were proposed in [39, 20], using the so-called “error invariants”. These are abstractions of the program state at each point that, combined with the rest of the trace, will imply the target. If the error invariant at two points is the same, the code between them is deemed irrelevant. The most important difference with our work is that these “error invariants” are not guaranteed to be loop invariants even at looping points, whereas the whole purpose of our work is to find loop invariants. This difference is because fundamentally, [39, 20] consider the trace simply as a sequence of transitions and are agnostic about loops.

Other related work include those that generate loop invariants, but may not be concerned with compressing execution traces. For instance, [13] uses a template-based invariant generation technique proposed in [26] to refine counter-example paths in the context of CEGAR [24]. Contrary to execution traces that terminate, CEGAR systems generate *abstract* counter-examples, and so [13] does not unroll loops, for it may go into non-termination. Other works that discover invariants using program analysis such as [52, 68] do not guarantee to find a safe invariant – a crucial requirement of trace compression. In our experiments, we used the tools [52] and [74] to generate invariants for some of our benchmarks, only to find that they did not return safe invariants.

Another work called Daikon by Ernst *et al.* [40] discovers “likely invariants” by instrumenting the program with predicates, executing test cases and checking which predicates are not falsified upto some sufficient degree of tests. These are reported to the programmer as being likely invariants at their respective program points. The main difference with our work is that we strive to produce *safe* invariants, whereas [40] only attempts to produce predicates that are “likely” to be invariants, but may or may not be safe. Having said that, their underlying invariant generation technique can still be utilised in our method to discover more sophisticated invariants. For instance, providing our BIID algorithm with likely predicates inferred by their method can help us confirm if the predicate is indeed an invariant. Another related work [45] presents several heuristics for computing invariants by mutating postconditions of loops. However their method performs no unrolling of loops in case the invariant was found to be



unsafe. Nevertheless, we believe their heuristics would be a good addition to our BIID algorithm to obtain better candidate invariants.

There are numerous other works such as [93, 41, 101, 75, 95] that discover invariants through static/dynamic analysis, testing, constraint solving, heuristics, etc. Invariant discovery is a heavily studied area for decades and so it is formidable to compare extensively with every technique proposed in the community. However, we make it clear that in our work, any invariant discovery method can be applied, i.e., our proposed BIID method can be augmented with any amount of sophistication to make our trace compression algorithm better. The main contribution of this work is not BIID itself, but an algorithm to *utilise* an invariant discovery technique such as BIID to compress traces, a novel backup mechanism if the discovery fails, and a method to provide an explanation of invariance.

Finally, our work is inspired by [59], which performs full symbolic execution of the program in the context of program verification. It however does not handle traces, and therefore has no way of checking if an invariant is safe until all loops in the program are compressed and the target is reached. Then, if the target is not implied, it needs a “refinement step” (similar to CEGAR) to find the unsafe invariant. Moreover, since it performs unbounded symbolic execution on the program, it is guaranteed to terminate. Nevertheless, our work can be considered an adaptation of [59] for traces. To the best of our knowledge, ours is first method to merge the two ideas: generate loop invariants to compress execution traces and provide a proof of invariance as an explanation.

Finally, we make it clear that there are some works such as [46, 76] that perform trace compression with an entirely different meaning. Working with a low level representation of a trace, their goals are to reduce its storage space, compression/decompression rate, and evaluating computer architecture (e.g., caches) with trace simulation. Clearly, we are completely unrelated to them except for the words “trace compression”.

## 6.2 Background

**Symbolic Execution.** We define symbolic execution in a slightly different manner than in Section 2.1 to suit the algorithm in this chapter. A *symbolic state*  $\mathfrak{v}$  is now defined as a tuple  $\langle \ell, C \rangle$ . The symbol  $\ell \in \Sigma$  corresponds to the current program location (with special symbols for initial location,  $\ell_{\text{start}}$ , and final location,  $\ell_{\text{end}}$ ).  $C$  is a set of constraints on the program variables at the location  $\ell$ , which is to be interpreted as a conjuncted first-order logic formula (e.g.,  $C = \{x > 5, y < 3\}$  is interpreted as the formula  $x > 5 \wedge y < 3$ ) that must be satisfied for symbolic execution to follow the particular corresponding path. The set of first-order formulas and symbolic states are denoted by *FOL* and *SymStates*, respectively.

Given a transition system  $\langle \Sigma, \longrightarrow \rangle$  and a state  $\mathfrak{v} \equiv \langle \ell, C \rangle \in \text{SymStates}$ , the symbolic execution of  $\ell \xrightarrow{\text{op}} \ell'$  returns another symbolic state defined as:

$$\text{SYMSTEP}(\mathfrak{v}, \ell \xrightarrow{\text{op}} \ell') \triangleq \begin{cases} \langle \ell', C \cup \{\bar{c}\} \rangle & \text{if op} \equiv \text{assume}(c) \text{ where } \bar{c} \text{ is } c \\ & \text{with proper renaming} \\ \langle \ell', C \cup \{x_k = \bar{e}\} \rangle & \text{if op} \equiv x = e \text{ where } x_k \text{ is fresh} \\ & \text{and } \bar{e} \text{ is } e \text{ with proper renaming} \end{cases} \quad (6.1)$$

Note that while adding the constraint to  $C$ , we rename each variable in the constraint to its latest version in  $C$ . For assignments, we create a fresh variable on the left hand side. This intuitively mimics a Static Single Assignment (SSA) based symbolic execution of the transition  $\ell \xrightarrow{\text{op}} \ell'$ .

Given a symbolic state  $\mathfrak{v} \equiv \langle \ell, C \rangle$  we define the *evaluation* of  $\mathfrak{v}$ , represented as  $\llbracket \mathfrak{v} \rrbracket : \text{SymStates} \rightarrow \text{FOL}$  as the *projection* of the constraints in  $C$  onto the set of program variables *Vars*. The projection is performed by eliminating existentially all auxiliary variables that are not in *Vars*. Intuitively  $\llbracket \mathfrak{v} \rrbracket$  is an FOL formula only on the latest versions of variables in  $C$ , and is equisatisfiable with  $C$ .

**Trace Semantics.** A *trace*  $\mathcal{T}$  is a sequence of transitions  $\ell_{\text{start}} \xrightarrow{\text{op}_1} \ell_1 \cdot \ell_1 \xrightarrow{\text{op}_2} \ell_2 \cdots \ell_{n-1} \xrightarrow{\text{op}_n} \ell_{\text{end}}$  such that each  $\ell_{i+1}$  is a successor of  $\ell_i$ , and is

obtained by executing the program  $\mathcal{P}$  with certain inputs. W.l.o.g, we assume that the inputs have been encoded as transitions in the trace itself. Abusing notation, we say “ $\mathfrak{v}$  is the symbolic state at  $\ell$  along the trace” if symbolic execution, starting with the state  $\mathfrak{v}_{\text{start}} \equiv \langle \ell_{\text{start}}, \emptyset \rangle$  at the beginning of the trace, results in the state  $\mathfrak{v}$  at  $\ell$  (if  $\ell$  is within a loop, there may be multiple symbolic states at  $\ell$ , in which case we would make unambiguous the state that we are referring to). Again w.l.o.g, we assume that a *target property* (e.g., an assertion) is provided at  $\ell_{\text{end}}$  that is implied by  $\mathcal{T}$ . We interpret this property as a FOL formula  $\phi$  on the program variables. Formally, if  $\mathfrak{v}_{\text{end}}$  is the symbolic state at  $\ell_{\text{end}}$  along the trace  $\mathcal{T}$ , then  $\llbracket \mathfrak{v}_{\text{end}} \rrbracket \models \phi$ .

A *compressed trace*  $\mathcal{T}_c$  is a sequence of transitions where some programs points are annotated with loop invariants. Formally,  $\mathcal{T}_c$  is a sequence  $\ell_{\text{start}} \xrightarrow{\text{op}_1} \ell_1 \cdot \ell_1 \xrightarrow{\text{op}_2} \ell_2 \cdots \ell_{m-1} \xrightarrow{\text{op}_m} \ell_{\text{end}}$  where there may exist transitions  $\ell_{\text{loop}} \xrightarrow{I, \mathcal{S}} \ell_{\text{loop}}$  where  $\ell_{\text{loop}}$  is a looping point,  $I$  is a FOL formula representing the loop invariant, and  $\mathcal{S}$  is the proof tree (essentially the symbolic execution tree) from which the Hoare-triples for the proof that  $I$  is an invariant will be extracted. An important property of  $\mathcal{T}_c$  is that the invariants are safe. That is, if  $\mathfrak{v}'_{\text{end}}$  is the symbolic state at  $\ell_{\text{end}}$  along  $\mathcal{T}_c$ , then  $\llbracket \mathfrak{v}_{\text{end}} \rrbracket \models \llbracket \mathfrak{v}'_{\text{end}} \rrbracket \models \phi$ . Moreover, the length of  $\mathcal{T}_c$  is at most the length of  $\mathcal{T}$  (i.e.,  $m \leq n$ ) as a result of looping points being compressed using invariants.

## 6.3 Algorithm

We now present our algorithm in two phases. In the first phase, we perform forward symbolic execution along the trace to compute inductive invariants to compress the loops. Here we attempt to discover the *strongest* possible invariants and also build the symbolic execution tree for each invariant discovered from which we will extract Hoare-triples. In the second phase, we generalise the invariants in the compressed trace obtained in phase one using backward weakest-precondition computation.

```

COMPRESSTRACE( $\mathcal{T}, \mathcal{P}, \phi$ )
1:    $\mathfrak{v} = \langle \ell_{\text{start}}, \emptyset \rangle$  and  $t = 1$ 
2:   while  $t \neq t_{\text{end}}$  do
3:     let  $\mathcal{T}[t] = \ell \xrightarrow{\text{op}} \ell'$ 
4:     if  $\ell$  is a loop from  $\mathcal{T}[t]$  to  $\mathcal{T}[t_{\text{exit}}]$  then
5:        $\langle \mathfrak{v}', \mathcal{S} \rangle = \text{LOOPINV\_FIXPO}(\mathfrak{v}, \mathcal{P})$ 
6:       if ( $\text{CHECKSAFEINV}(\mathfrak{v}', \mathcal{T}, t_{\text{exit}}, \phi)$ ) then
7:          $t = t_{\text{exit}} + 1$ 
8:          $\mathcal{T}_c = \mathcal{T}_c \cdot \ell \xrightarrow{\llbracket \mathfrak{v}' \rrbracket, \mathcal{S}} \ell$ 
9:          $\mathfrak{v} = \langle \ell_{\text{exit}}, C' \rangle$  where  $C'$  is the constraint list
           of  $\mathfrak{v}'$  and  $\ell_{\text{exit}}$  is the loop exit point
10:        continue
11:      endif
12:    endif
13:     $\mathcal{T}_c = \mathcal{T}_c \cdot \ell \xrightarrow{\text{op}} \ell'$ 
14:     $\mathfrak{v} = \text{SYMSTEP}(\mathfrak{v}, \ell \xrightarrow{\text{op}} \ell')$  and  $t = t + 1$ 
15:  end
16:  return  $\text{GENERALISE}(\mathcal{T}_c, \mathcal{P}, \phi)$ 

CHECKSAFEINV( $\mathfrak{v}, \mathcal{T}, t_{\text{exit}}, \phi$ )
17:  for  $t = t_{\text{exit}}$  to  $t_{\text{end}}$  do
18:    let  $\mathcal{T}[t] \equiv \ell \xrightarrow{\text{op}} \ell'$ 
19:     $\mathfrak{v} = \text{SYMSTEP}(\mathfrak{v}, \ell \xrightarrow{\text{op}} \ell')$ 
20:  end
21:  if  $\llbracket \mathfrak{v} \rrbracket \models \phi$  then return true else return false

```

Figure 6.3: Loop Compression with Invariants

### 6.3.1 Loop compression with invariants

Our main algorithm consists of the procedures shown in Fig. 6.3. The main procedure, COMPRESSTRACE, takes as inputs the trace  $\mathcal{T}$ , program  $\mathcal{P}$  and the target  $\phi$ , and returns a compressed trace  $\mathcal{T}_c$  as defined in Section 6.2. It models the trace  $\mathcal{T}$  as an array using the variable  $t$  as the index variable, and  $t_{\text{end}}$  its length.

It starts by initialising  $\mathfrak{v}$ , representing the current symbolic state, to  $\langle \ell_{\text{start}}, \emptyset \rangle$  and  $t$  to 1. In line 2, a loop runs till the end of the trace is reached (i.e., till  $t$  becomes  $t_{\text{end}}$ ), doing the following in each iteration. Assuming that the current transition along the trace is from  $\ell$  to  $\ell'$ , it checks if  $\ell$  is the starting point of a loop. If so, then let the loop's iterations in  $\mathcal{T}$  run from the current trace index  $t$

to, say,  $t_{\text{exit}}$ . That is,  $\mathcal{T}[t]$  is the transition from  $\ell$  to the loop body and  $\mathcal{T}[t_{\text{exit}}]$  is the transition from  $\ell$  to the loop's exit.

Now the algorithm attempts to discover an invariant for this loop in the program by calling the procedure `LOOPINV_FIXPO` with the current symbolic state  $\mathfrak{v}$  and the program  $\mathcal{P}$  (line 5). In principle, this procedure can implement any algorithm that generates a loop invariant, for example [13, 26, 74]. We only require the procedure to return a tuple  $\langle \mathfrak{v}', \mathcal{S} \rangle$  where  $\mathfrak{v}'$  is an invariant state at  $\ell$  (i.e.,  $\llbracket \mathfrak{v} \rrbracket \models \llbracket \mathfrak{v}' \rrbracket$  and  $\llbracket \mathfrak{v}' \rrbracket$  is an invariant) and  $\mathcal{S}$  is some kind of proof that  $\llbracket \mathfrak{v}' \rrbracket$  is indeed invariant through the loop at  $\ell$ . In this work, this procedure will implement the BIID technique, which is one particular way of discovering invariants using symbolic execution (SE) and using the SE tree as the proof tree for invariance.

Now that  $\mathfrak{v}'$  is an invariant at  $\ell$ , the algorithm then checks whether it is a *safe invariant*. The idea is to symbolically execute the trace after the loop with the discovered invariant as the state and to check if the target is implied at the end. This is done by calling `CHECKSAFEINV` at line 6 with the invariant state  $\mathfrak{v}'$ , the trace  $\mathcal{T}$  and  $t_{\text{exit}}$ , the index of the loop's exit transition along  $\mathcal{T}$ , and the target  $\phi$ . `CHECKSAFEINV` basically implements the symbolic execution along the trace starting at  $t_{\text{exit}}$  till  $t_{\text{end}}$  (lines 17-20), and checks whether the symbolic state at  $t_{\text{end}}$  implies  $\phi$  (line 21).

If the check passed, then  $\mathfrak{v}'$  represents a safe invariant, meaning the loop has been compressed. Therefore we can continue with our method along the rest of the trace after the loop. Recall that  $t_{\text{exit}}$  was the index in  $\mathcal{T}$  for the loop's exit transition. Hence, in line 7, the algorithm assigns the trace index variable  $t$  to  $t_{\text{exit}}+1$ . In the next line, it records the safe invariant in the compressed trace  $\mathcal{T}_c$  by adding to it the (looping) transition  $\ell \xrightarrow{\llbracket \mathfrak{v}' \rrbracket, \mathcal{S}} \ell$ , where  $\llbracket \mathfrak{v}' \rrbracket$  is the invariant, and  $\mathcal{S}$  is the proof for its invariance. Finally, the symbolic state  $\mathfrak{v}$  is updated to  $\langle \ell_{\text{exit}}, \mathcal{C}' \rangle$  to signify that symbolic execution should continue from the loop's exit point  $\ell_{\text{exit}}$  with the constraint list  $\mathcal{C}'$  that carries the invariant's ( $\mathfrak{v}'$ ) state.

If the check at line 6 failed, it means the invariant turned out to be unsafe, in which case our algorithm discards the work done, and unrolls the loop by simply following the trace  $\mathcal{T}$ . It adds the current transition from  $\ell$  to  $\ell'$  to the

compressed trace  $\mathcal{T}_c$  (line 13), and symbolically executes the current state  $\mathfrak{v}$  with the current transition to get the next state (line 14). These steps simulate unrolling the loop along the trace  $\mathcal{T}$ , until a loop header is reached again at line 4. The entire process continues until the end of the trace  $\mathcal{T}$  is reached, at which point it calls `GENERALISE` to generalise the invariants in  $\mathcal{T}_c$ .

### **Basic Individually Invariant Discovery (BIID)**

We now present our particular method to discover invariants and their respective proofs when calling `LOOPINV_FIXPO`. At a high level, our method follows paths in the transition system  $\mathcal{P}$  starting at the given loop header, and at the end of each path  $\pi$  that reaches the looping point again (making a cycle), it deletes individual constraints at the loop header that do not still hold at the end of  $\pi$ . It then backtracks and explores all symbolic paths in the loop.

The entire process is repeated until no deletions are made (i.e., fixpoint is reached). This entails that the constraints left undeleted at the loop header still hold at the end of every path through the loop, in other words, being invariant through the loop. The symbolic execution tree generated at fixpoint provides the proof of invariance. This method provides a fine a balance between getting the strongest invariants and efficiency.

`LOOPINV_FIXPO` (Fig. 6.4) is a wrapper procedure that implements the fixpoint computation (lines 1-4) on the symbolic state  $\mathfrak{v}$ . It initialises  $\pi$  to  $\emptyset$  (the empty sequence) and  $\mathcal{S}$  to  $\emptyset$ , where  $\pi$  will be used by another procedure to represent the current symbolic path, and  $\mathcal{S}$  will eventually represent the symbolic execution tree from which the proof that  $\llbracket \mathfrak{v} \rrbracket$  is an invariant is extracted. It then calls the procedure `LOOPINV` passing the state  $\mathfrak{v}$  and these initialised variables. This process is repeated until fixpoint is reached (line 4).

`LOOPINV` is a recursive procedure that symbolically explores all paths in the loop from the current symbolic state  $\mathfrak{v}$ . In lines 9-13, for each transition from  $\ell$  in the program  $\mathcal{P}$ , it first obtains the next state  $\mathfrak{v}'$  by performing a symbolic step from  $\mathfrak{v}$ . Then, if  $\mathfrak{v}'$  is not an infeasible state, it recursively calls itself with  $\mathfrak{v}'$ , appending  $\mathfrak{v}$  to the current path  $\pi$  to signify that it has been reached. The symbolic tree returned, corresponding to the execution of  $\mathfrak{v}'$ , is stored in

```

LOOPINV_FIXPO( $v, \mathcal{P}$ )
1:  do
2:       $v'' = v$  and  $\pi = \emptyset$  and  $\mathcal{S} = \emptyset$ 
3:       $\langle v, \mathcal{S} \rangle = \text{LOOPINV}(v, \mathcal{P}, \pi, \mathcal{S})$ 
4:  until  $v'' == v$ 
5:  return  $\langle v, \mathcal{S} \rangle$ 

LOOPINV( $v \equiv \langle \ell, C \rangle, \mathcal{P}, \pi, \mathcal{S}$ )
6:  if  $\exists v_h \equiv \langle \ell, \cdot \rangle \in \pi$  then
7:      REMOVE_NONINV( $v_h, v$ )
8:       $\mathcal{S} = \mathcal{S} \cup \{\pi\}$  and return  $\langle v, \mathcal{S} \rangle$ 
9:  foreach  $\ell \xrightarrow{\text{op}} \ell' \in \mathcal{P}$  do
10:      $v' = \text{SYMSTEP}(v, \ell \xrightarrow{\text{op}} \ell')$ 
11:     if  $\llbracket v' \rrbracket$  is unsat then continue
12:      $\langle \cdot, \mathcal{S}' \rangle = \text{LOOPINV}(v', \mathcal{P}, \pi \cdot v, \mathcal{S})$ 
13:      $\mathcal{S} = \mathcal{S} \cup \mathcal{S}'$ 
14:  end
15:  return  $\langle v, \mathcal{S} \rangle$ 

REMOVE_NONINV( $v_h \equiv \langle \ell, C \rangle, v$ )
16:  let  $\llbracket v_h \rrbracket$  be  $c_1 \wedge c_2 \wedge \dots \wedge c_n$ 
17:  foreach  $c_i$  in  $\llbracket v_h \rrbracket$  do
18:     if  $\llbracket v \rrbracket \not\models c_i$  then  $C = C \setminus \{c_i\}$ 
19:  end

```

Figure 6.4: Basic Individually Invariant Discovery

$\mathcal{S}'$  which is then combined together with  $\mathcal{S}$  (line 13). In lines 6-8, it checks if a cyclic looping point has been reached (i.e., the current program point  $\ell$  has already been visited along the path  $\pi$ ). If so, we need to remove constraints from the loop header that were not invariant through  $\pi$ . This is done by calling REMOVE\_NONINV with  $v_h$ , the symbolic state at the loop header. Then, since the end of the path has been reached, it simply adds  $\pi$  to the symbolic tree  $\mathcal{S}$  and returns (line 8).

REMOVE\_NONINV is a straightforward procedure. It first obtains the list of constraints at  $v_h$  in evaluated form by applying  $\llbracket v_h \rrbracket$  (line 16). Then, for each constraint  $c_i$  it checks if  $c_i$  still holds at the end of the path by checking if  $\llbracket v \rrbracket$  entails  $c_i$ . If not,  $c_i$  is deleted from the list of constraints at  $v_h$  (lines 17-19).

At the highest level LOOPINV\_FIXPO repeatedly calls LOOPINV until no more deletions are made at the loop header in REMOVE\_NONINV. Once fixpoint

is reached (line 4),  $\nu$  becomes an invariant state at the loop header. It is then returned along with  $\mathcal{S}$  which serves as the proof that  $\nu$  is indeed invariant.

**Important note about complexity.** In general, there can be an exponential number of paths in a loop. Since our algorithm naively explores all paths, it can easily become intractable. This problem, called “path-explosion”, is well-known in symbolic execution and is tackled using *interpolation* [66, 83, 63] (see Chapter 2 Definition 1). The idea is to avoid the (redundant) exploration of a symbolic state  $\nu$  at a program point  $\ell$  if it is found to be *equivalent* to another state  $\nu'$  at  $\ell$  (i.e.,  $\llbracket \nu \rrbracket = \llbracket \nu' \rrbracket$ ). Interpolation increases the chances of this happening by discarding certain irrelevant information when comparing  $\nu$  and  $\nu'$ .

In other words, when  $\nu'$  was explored, interpolation would remove certain constraints from it such that even if  $\llbracket \nu \rrbracket \models \llbracket \nu' \rrbracket$  (a weaker condition than before),  $\nu$  can be considered equivalent to  $\nu'$ , and need not be explored. Although interpolation itself is orthogonal to this particular chapter, it is an important optimisation without which our algorithm cannot scale. We tacitly assume that if interpolation is used to build the symbolic tree, symbolic states are interpolated and merged when they need not be redundantly explored.

### 6.3.2 Invariant Generalisation

The idea in this phase is to perform a (backward) *weakest precondition* computation along the compressed trace starting from the target, and to logically weaken the invariants as long as the weakest precondition is implied. This algorithm is shown in Fig. 6.5.

The GENERALISE procedure takes as input the compressed trace  $\mathcal{T}_c$  along with the target  $\phi$  and the program. It begins by initialising the “post-condition” variable  $\Psi$  to  $\phi$  at line 1. Then, starting at  $t_{c_{\text{end}}}$  (the end of  $\mathcal{T}_c$ ) and going backwards, it does the following at each step. If the current transition is not an annotated loop invariant, it computes the weakest liberal precondition ( $\widehat{wlp}$ ) of  $\Psi$  along the transition  $\text{op}$  (lines 3-4). The  $\widehat{wlp}$  is defined as the weakest formula on the pre-state such that the execution of  $\text{op}$  results in the post-state  $\Psi$ , modulo termination. In practice, it can be approximated by making a linear number of



```

GENERALISE( $\mathcal{T}_c, \mathcal{P}, \phi$ )
1:    $\Psi = \phi$ 
2:   for  $t_c = t_{c_{\text{end}}}$  to 1 do
3:     if  $\mathcal{T}_c[t_c] \equiv \ell_1 \xrightarrow{\text{op}} \ell_2$  then
4:        $\Psi = \widehat{wlp}(\Psi, \text{op})$ 
5:     else if  $\mathcal{T}_c[t_c] \equiv \ell \xrightarrow{I, \mathcal{S}} \ell$  then
6:        $I' = \nabla(I, \Psi)$ 
7:        $v' = \langle \ell, I' \rangle$ 
8:       if LOOPINV_FIXPO( $v', \mathcal{P}$ ) returns  $\langle v'', \mathcal{S}' \rangle$ 
          s.t.  $v'' = v'$  then
9:          $\mathcal{T}_c[t_c] = \ell \xrightarrow{I', \mathcal{S}'} \ell$ 
10:         $\Psi = I'$ 
11:      else  $\Psi = I$ 
12:   end
13:   return  $\mathcal{T}_c$ 

```

Figure 6.5: Invariant Generalisation using Weakest Precondition

calls to a theorem prover using techniques outlined in [66].

If the current transition is a loop invariant annotation (line 5)  $I, \mathcal{S}$  at  $\ell$ , it first computes a *widening* of the invariant  $I$  w.r.t. the post-condition  $\Psi$ . The widening operator  $\nabla$  returns a formula  $I'$  such that  $I \models I' \models \Psi$ , i.e.,  $I'$  is weaker than  $I$  but still strong enough to imply the post-condition. However, we need to check if the weakened formula is still invariant through the loop. This is done in lines 7-8 by calling LOOPINV\_FIXPO and checking whether it returns the same invariant state  $v'$  at the loop header. If so, then line 9 replaces the current invariant annotation in  $\mathcal{T}_c$  with the new annotation  $I', \mathcal{S}'$  where  $\mathcal{S}'$  is the symbolic tree generated for  $I'$  by LOOPINV\_FIXPO (note that  $\mathcal{S}'$  can be different from  $\mathcal{S}$  because it is a proof tree for a different, weaker, invariant). Finally, line 10 sets  $I'$  to be the post-condition that is propagated backward.

If the weakened formula  $I'$  was not invariant, the algorithm makes no change to the existing annotation and simply propagates  $I$  backward (line 11). Once the beginning of the compressed trace is reached, the algorithm returns  $\mathcal{T}_c$  – the compressed trace containing now generalised invariants with proofs.



$$4) \{i = 0, j \geq 1\} \quad x=1 \quad \{i = 0\}$$

$$5) \{i = 0, j < 1\} \quad x=-1 \quad \{i = 0\}$$

Applying the rule for conditional statements on triples 4 and 5:

$$6) \{i = 0\} \quad \mathbf{if} (j \geq 1) \quad x=1 \quad \mathbf{else} \quad x=-1 \quad \{i = 0\}$$

Applying the rule for composition on triples 6 and 3 we get:

$$7) \{i = 0\} \quad \mathbf{if} (j \geq 1) \quad x=1 \quad \mathbf{else} \quad x=-1 \\ \quad \quad \quad j=j+1 \quad \quad \quad \{i = 0\}$$

Along with the loop condition, we get the triple:

$$8) \{i = 0, \quad \mathbf{if} (j \geq 1) \quad x=1 \quad \mathbf{else} \quad x=-1 \\ \quad \quad \quad j < 10\} \quad j=j+1 \quad \quad \quad \{i = 0\}$$

Finally, applying the rule for loops on triple 8 we get:

$$9) \{i = 0\} \quad \mathbf{while} (j < 10) \\ \quad \quad \quad \mathbf{if} (j \geq 1) \quad x=1 \quad \mathbf{else} \quad x=-1 \\ \quad \quad \quad j=j+1 \quad \quad \quad \{i = 0\}$$

which proves the invariance of  $\{i = 0\}$  through the loop.

The basic idea is of course that our Hoare proof is often more concise than the original trace, thus enhancing understandability. There is however an additional observation that emphasises this: that many triples are in fact *trivial* because their validity follows from the *frame rule*. Informally, this means any triple  $\{P\}S\{P\}$ , where the precondition is the same as the postcondition, holds if all write operations in  $S$  can be shown to be separate from the “footprint” of  $P$ . This footprint is, informally, the set of all stack and heap locations which influence the truth value of  $P$ . This observation is widely used and formalised in Separation Logic [89].

For instance, in the above proof, all triples are in fact trivial because all statements neither affect the fact that  $i = 0$  nor modify  $i$ . As another example, in Fig. 6.6(c), the triple

$$\{i \geq 1, x \geq 0, y \geq 0\} \quad k=k+1 \quad \{i \geq 1, x \geq 0, y \geq 0\}$$

is trivial because the pre- and post-conditions are the same and none of the variables in them are modified by the statement  $k=k+1$ .

For our purposes, we will show in our experimental evaluation that not only is the total number of triples significantly smaller than the length of the original

trace (that is, significant compression is achieved), but further, that a large proportion of the triples are in fact trivial. This then further strengthens the position that our Hoare proofs are more understandable.

## 6.4 Experimental Evaluation

Benchmark	Trace length		%C	#Triples (trivial)	#U	Time
	Orig.	Com.				
SSH client	462	95	80%	47(43)	6	289s
SSH server	346	13	96%	91(84)	0	94s
tokenring	885	218	75%	66(37)	2(2)	161s
cdaudio	1434	121	92%	15(11)	0	128s
floppy	398	83	80%	4(2)	1	2s

Table 6.1: Trace statistics for our experiments. %C: percentage compression, #U: number of unrolls until compression was achieved (inner loop unrolls, if any)

We implemented our algorithm on the TRACER [64] framework for symbolic execution and evaluated it on several medium-sized benchmarks from the software verification competition (SV-COMP 2013). The programs are all unsafe, and in order to work with a meaningful trace, we invoked Directed Automated Random Testing (DART) [49], commonly known as *concolic testing*, to obtain inputs that caused the safety property to be violated. We then grounded the inputs into the program, and set our target for the resulting “error trace” to be the negation of the safety property. We also implemented slackening [66] in BIID to get more candidates to test for invariance. To make the experiments simpler we applied static slicing (provided by Frama-C [1]) to remove statically irrelevant statements. All experiments were run on an Intel 2.3Ghz system with 2GB memory. We first tabulate the results in Table 6.1 for convenience, and explain each benchmark in detail.

### 6.4.1 SSH Client (`s3_clnt_1_false.cil.c`)

Our first example is a buggy SSH client program from the ssh-simplified suite. A simplified view of the program is shown in Fig. 6.7. It consists of a big loop that reads the current state variable `s` and performs some action, then setting `s`

SSH Client		
Program	Trace (len: 462)	Compressed (len: 95)
<pre> end=flag=err=0 s=A while (end==0) do   if (s==A)     if (flag==0)       flag=1       s=B     else if (s==B)       if (flag==1)         flag=2         s=C       ...     else if (s==l)       if (flag==4)         err=1         s=J       else if (s==J)         s=K       ...     else end=1 end <b>TARGET:</b> {err = 1} </pre>	<pre> end=flag=err=0 s=A end==0   s==A     flag==0     flag=1     s=B   end==0   s==B     flag==1     flag=2     s==C   ... end==0   s==l     flag==4     err=1   s=J end==0   s==J   s=K end==0 ... end=1 end≠0 <b>TARGET:</b> {err = 1} </pre>	<pre> end=flag=err=0 s=A end==0   s==A     flag==0     flag=1     s=B   end==0   s==B     flag==1     flag=2     s==C   ... end==0   s==l     flag==4     err=1   s=J <b>INV:</b> {err = 1,         flag = 4} end≠0 <b>TARGET:</b> {err = 1} </pre>

Figure 6.7: The SSH client program, the error trace and the compressed trace

to the next state (A, B, etc. are the states). In certain states, a flag variable is checked to be of some value, and is set to the next value. In a particular state l, if flag was found to be 4, the error variable err is set to 1, and flag is not modified thereafter.

We set the target to the negation of the safety property, namely,  $\{err = 1\}$ . On running the code with buggy inputs from concolic testing, the trace iterated through the loop 40 times, executing a total of 462 transitions, and implied the target as shown under Trace in Fig. 6.7. It is noteworthy that dynamic slicing was unable to remove any loop iteration as a whole, as the state s changes in each iteration and the loop's exit is (control) dependent on s being a certain value.

In order to compare with other invariant discovery methods, we encoded the inputs into the program itself and computed invariants using (1) the polyhedra abstract domain of APRON [68] and (2) the INVGEN [52] tool. Both of them returned the *unsafe* invariant  $\{\mathbf{flag} \geq 0, 0 \leq \mathbf{end} \leq 1\}$ .

Our algorithm initially only discovered the unsafe invariant *true*. After one unroll, it again discovered an unsafe invariant that the initial state was A. Then, after **six(6)** unrolls, the trace reached the state I, checked if `flag` was 4, and set `err` to 1 and the next state to J. At this point, our algorithm discovered the *safe* invariant  $\{\mathbf{err}=1, \mathbf{flag}=4\}$ , that implies the target  $\{err = 1\}$ , as shown on the right of Fig. 6.7. Note that  $flag = 4$  is needed to preserve the invariance of  $err = 1$ . This is a practical example of a loop converging towards stronger invariants as it iterates, culminating in a safe invariant.

Ultimately, the trace was reduced from **462** transitions to **95** transitions (**80%** compression). Our algorithm also discovered 3 invariants on a few other variables in the program, but the generalisation phase deleted them as they were irrelevant to the target. The proof for the invariant contained **47** triples, of which **43** were classified as “trivial” according to the frame-rule in Section 6.3.3. The rest **4** of the triples are the ones non-trivially contributing to the explanation of the invariant (the **4** triples correspond to the specific case I where the error occurred). The process took **289s** to complete.

#### 6.4.2 SSH Server (`s3_srvr_6_false.cil.c`)

Our next benchmark is a buggy SSH server program from the `ssh-simplified` suite. The structure of this program is similar to the SSH Client benchmark, in that a main loop iterates using a state variable `s` until it reaches a particular value. There are two safety properties: one outside and one inside the loop, as shown in Fig. 6.8 under **SSH Server**. As before the target is  $\{err = 1\}$ . On running the buggy inputs, the trace executed the loop **14** times before implying the target, executing a total of **346** transitions. Also, both external invariant discovery methods – the polyhedra abstract domain of APRON, and the INVGEN tool computed the unsound invariant *false* as they somehow interpreted the loop to be never executed at all.

SSH Server		
Program	Trace (len: 346)	Compressed (len: 13)
<pre> end=flag=err=0 <b>if</b> (s_info_cb!=0)   cb=s_info_cb <b>else</b> {   <b>if</b> (s_ctx_cb!=0)     cb=s_ctx_cb   <b>if</b> (cb!=0)     err=1 } <b>while</b> (end==0) <b>do</b>   <b>if</b> (s==A)     <b>if</b> (flag==0)       flag=1     s=B     ...   <b>else if</b> (s==I)     <b>if</b> (flag==4)       err=1     s=J     ...   <b>else</b> end=1 <b>end</b> <b>TARGET:</b> {err = 1} </pre>	<pre> Input: <b>s_info_cb = 0</b> <b>s_ctx_cb = -1</b> end=flag=err=0 s_info_cb == 0 s_ctx_cb != 0 cb=s_ctx_cb cb != 0 err=1 end==0 s==A flag==0 flag=1 s=B end==0 ... s==I flag==4 err=1 ... end=1 end≠0 <b>TARGET:</b> {err = 1} </pre>	<pre> Input: <b>s_info_cb = 0</b> <b>s_ctx_cb = -1</b> end=flag=err=0 s_info_cb == 0 s_ctx_cb!=0 cb=s_ctx_cb cb!=0 err=1 <b>INV:</b> {err = 1} end ≠ 0 <b>TARGET:</b> {err = 1} </pre>

Figure 6.8: The SSH server program, the error trace and the compressed trace

On invoking our algorithm, we reached the loop header with the symbolic state that included  $\{err = 1\}$  because the safety property before the loop was itself violated. On symbolically exploring the loop, we concluded that even though `err` is modified in the loop, this constraint on `err` remains invariant, which happened to be safe. Thus, we were able to compress the loop even without any unrolling. The resulting compressed trace contained only **13** transitions and **91** triples, of which **84** were classified trivial. The **7** non-trivial triples correspond to the case where the error occurred in the loop. We also discovered 20 other invariant constraints which were removed by the generalisation phase. The entire process took **94s** to complete.

### 6.4.3 Tokenring (token\_ring01\_unsafe.c)

Our third benchmark is a buggy token ring algorithm from the loops suite. We do not show the program or its trace due to its large size, but give an overview. It consists of two functions `master` and `transmit`, and in the main function, a loop is executed calling (non-deterministically) these two functions in each iteration. This loop is nested within another loop that keeps re-running the whole simulation for a given number of times. In a certain simulation of the inner loop, when the two functions are called in a particular sequence, an error is triggered in `master`.

The inputs in this program controlled the number of iterations of the inner and outer loops, and determined the specific combination of calls that triggers the error. The inputs directed the trace to execute **5** iterations of the outer loop, and for each of those, **6** iterations of the inner loop – a total of **30** iterations that resulted in **885** transitions. We were unable to invoke the two external invariant generators on this benchmark (as well as the next two `cdaudio` and `floppy`) as their C front-end does not support programs with function calls.

On applying our method, it was unable to find an invariant for the outer loop immediately, so it began unrolling it. However, for the first iteration of the outer loop, it was able to find an invariant for the *inner* loop and compress it. In the second iteration, it in turn unrolled the inner loop twice to find a safe invariant and compress it, which then compressed the outer loop as well. This benchmark exhibited a remarkable feature of our method in practice – even if we are unable to compress an outer loop iteration, we can still compress the inner loop in that iteration.

Ultimately, the trace was reduced from **5** outer loop iterations, each containing **6** inner loop iterations, to just **2** outer loop iterations, each containing a *compression* of the inner loop, and finally a compression of the outer loop. The number of transitions was reduced from **885** to **218** (**75%** compression). Importantly, this compressed trace shows that the particular sequence of calls that triggers the error happened only in the second simulation (i.e., second outer loop iteration). This is quite valuable information to a programmer as he/she can quickly focus debugging efforts on that part, rather than checking which sim-



Bench -mark	Bound	Trace length		%C	#U	Time
		Orig.	Compr.			
cdaudio	4	342	121	65%	0	97s
	8	498	121	75%	0	99s
	16	810	121	85%	0	106s
	32	1434	121	92%	0	128s
floppy	4	146	83	43%	1	2s
	8	182	83	54%	1	2s
	16	254	83	67%	1	2s
	32	398	83	80%	1	2s

Table 6.2: Trend with varying loop bounds for cdaudio and floppy

ulation caused the error. Finally, **66** triples were generated, of which **37** were classified as trivial, and the rest **29** as non-trivial. The entire process took **161s** to complete.

#### 6.4.4 cdaudio (cdaudio\_simpl1\_unsafe.cil.c)

Our fourth benchmark is the buggy version of the Windows NT Driver “cdaudio” from the ntdrivers-simplified suite. It consists of about 15 safety properties of which exactly 1 is violated (i.e., the program is safe without that property). Concolic testing generated inputs that violated this property after going through the (only) loop in the program. We noticed that the loop has an *arbitrary* bound, i.e., the bound is simply the number of attempts made to start the CD-device, set to 4 by default. Once the device is started, signified by a `status` variable being a certain value, the loop exits.

Within the loop, there exist statements modifying variables that appear in the safety of other properties in the program. However, these properties are not affected because of two reasons: (1) the properties capture the *relationship* between the variables and not the actual value (e.g., the property  $s == NP$ , where  $s$  is set to  $NP$  in the loop) and (2) given the incoming context to the loop, many of these statements are along *infeasible paths*. Executing the program with the default bound of 4, the trace ran through **342** statements. On invoking our algorithm, we were able to compress all 4 iterations of the loop with safe invariants that either captured the exact relationship between the variables (e.g.,  $s == NP$ ) or their values which are sufficient to establish the relationship (e.g.,  $s == 1$  and

NP==1). The trace was reduced to **121** statements, and **15** triples were produced, of which **11** were classified trivial.

Since the bound is arbitrary, we tried to increase it in order to see a *trend* of our compression method. In Table 6.2, the row `cdaudio` shows the statistics for bounds 4, 8, 16 and 32 for the loop. In all cases our compression resulted in the trace being **121** transitions long. That is, even if the loop bound is increased, we were able to compress it using the same invariant without additional unrolling. In all cases, **15** triples were generated as before, out of which **11** were trivial. As it can be seen, the amount of compression approaches more than **90%** as the number of iterations increases. Moreover, the timing is not affected drastically, as we were able to finish in about **2 minutes** in all cases. The slight increase in timing is due to the `CHECKSAFEINV` procedure that has to run along the now longer trace to check if a discovered invariant is safe.

#### 6.4.5 floppy (`floppy_simpl3_unsafe.cil.c`)

Our final benchmark is another buggy Windows NT Driver “floppy” from the `ntdrivers-simplified` suite. Similar to `cdaudio`, this program also has about 20 safety properties out of which exactly 1 causes the program to be unsafe. There is one loop in the program that among other things, assigns either 0 or a negative value, say  $N$ , to a variable `ntStatus` non-deterministically. If `ntStatus` is assigned 0, or the loop’s bound is exhausted, the loop exits. If it is never assigned 0, its value is passed across many functions to a variable in the main function called `status`, which is then checked to be equal to 259. If it is not, the error is triggered.

On generating inputs that exercise the error trace, we set the target to be  $\{status = N\}$ . We again noted that the loop bound is arbitrary, and in fact, not even specified in the program. Hence we experimented with varying bounds. Due to lack of a “default” bound, we simply used the loop bounds from `cdaudio`, i.e., 4, 8, 16 and 32. Again, dynamic slicing was unable to remove any iteration as the safety variable `status` depends on `ntStatus` which is modified for every iteration within the loop, whose exit is in turn control-dependent on the assignment to `ntStatus`.

When our algorithm was invoked on the trace, we were unable to discover a safe invariant right away, as the loop destroys the initial value of `ntStatus` (i.e., 0) by setting it to  $N$  thereby preventing our invariant to capture any constraint on `ntStatus`. After one unroll however, we were able to capture the constraint  $\{ntStatus = N\}$  which is now invariant through the loop. This turned out to be a safe invariant, compressing the remaining iterations of the loop.

As before, we show in Table 6.2 the compression trend for this benchmark. We obtained traces of sizes **146** to **398** by varying the bounds. In all cases however, we were able to compress the trace to **83** transitions after **1** unroll. The amount of compression varies between **43%** to **80%** depending on the bound. In all cases, **4** triples were generated out of which **2** were classified trivial. The **2** non-trivial triples explain that `ntStatus` is assigned the value  $N$  within the loop. We were able to finish compressing the traces very quickly, within **2 seconds**.

## 6.5 Summary

We presented a novel method that combines loop invariant discovery with trace compression. We discussed major challenges that face this seemingly simple idea – (1) ensuring that invariants are safe, (2) a backup measure in case a safe invariant could not be found, (3) generalising safe invariants to aid in understanding, and (4) generating a proof of invariance as an explanation. We showed our method works well in practice by evaluating it on real-life benchmarks. We believe this would be one more step towards helping programmers with their daily practical problem of debugging.

# Chapter 7

## Conclusion

In this thesis we studied the application of symbolic execution to various program reasoning problems. We began by motivating the need for *path-sensitivity*, the art of analysing programs paying heed to the feasibility of paths, so that spurious information about the program from infeasible paths is excluded. We showed that symbolic execution is a powerful and versatile technique for path-sensitive program reasoning, owing to its “path condition” that forms the core of its flexibility.

We then looked at one of the major challenges faced by symbolic execution and path-sensitive analyses in general—the “path explosion” problem. This problem exists because of considering, in general, the feasibility of an *exponential* number of paths in a program. To address this problem, we made use of two techniques: *interpolation*, which has been employed recently to mitigate state space blowup in model checking, and the concept of *witnesses*. The idea is to alleviate path explosion during symbolic execution by opportunistically *merging* several symbolic states into one provided certain conditions—dictated by interpolants and witnesses—are met. We then put forward perhaps the most important theorem of the thesis (Theorem 1) that stated that no analysis information is lost during a merge that is governed by these two conditions.

Having established the above path-sensitive analysis framework with symbolic execution, we addressed four specific problems faced often by programmers: program slicing, testing, verification and trace understanding. We briefly summarise our contributions below.

In the area of backward slicing, our contribution is two-fold. First, we formulated an algorithm for efficient path-sensitive *static* slicing, which guarantees to produce “exact” slices for loop-free programs, and limited only by general loop invariant discovery technology for programs with loops. Second, we extended this slicing technique beyond static slicing, introducing a program transformation method based on the notion of *tree slicing*. We showed how tree slicing can be more powerful in reducing the program’s search space for external applications such as testing and verification.

In the area of testing, our contribution is to bring the machinery of interpolation and subsumption to concolic testing for the first time. We presented the framework of full and half interpolants, and showed how in concolic testing, the *modus operandi* of interpolation fails to provide benefit. We then proposed a novel technique of greedily accelerating the formation of interpolants during the testing process, that brings back the exponential benefits typical of interpolation.

In the area of verification, we first examined the problem with symbolic execution’s inherent avoidance of infeasible paths, a characteristic that degrades the quality of the discovered interpolants, thereby hindering subsumption in future. We then proposed a novel “lazy” strategy for symbolic execution that ignores these infeasibilities in pursuit of better interpolants. Although sounding contradictory to the principle of symbolic execution—which by default “eagerly” detects and avoids infeasibilities—we showed that this lazy version outperformed its canonical counterpart substantially in verification.

In the relatively new area of trace understanding, our contribution is a new algorithm to compress and explain error traces by discovering *safe loop invariants* for loop iterations in the trace. We showed that our algorithm produces significantly smaller traces, with loops replaced by a concise “explanation” in the form of Hoare-triples that prove that the discovered safe invariant for the loop is indeed one. Our algorithm is based on the principle that loops typically converge towards an invariant as they iterate, and we showed that this holds in practice by discovering safe invariants for several programs after a few unrolls.

Finally, we demonstrated our implementation of symbolic execution with interpolation—the TRACER framework, which forms the basis of this work.

## 7.1 Future Directions

There are two other major areas of application for our PSS-CFG, presented in Part II of Chapter 3, namely program analysis and execution. In analysis, the use of target variables is limited, or more often, nonexistent. Instead, the PSS-CFG provides a path-sensitive representation for *insensitive* (or limitedly sensitive) analysers to take advantage of the lack of infeasible paths to increase analysis precision. In program execution, the idea is, as in partial evaluation, to run a specialised version of the program depending on the target variables. This would always be faster than executing the general original program. We in fact have some early experimental results, not presented here. For analysis, we considered the (insensitive) alias analysis tool Crystal [91] on several programs and consistently noticed accuracy gains. For execution timing, we noticed, again on several programs, that the number of executed program steps was significantly reduced when using the PSS-CFG over the original program.

We are also conducting research (ongoing) [84] in the area of error trace explanation and localisation using interpolants. Recently, two different approaches have emerged for error trace explanation—one based on minimal unsatisfiable cores [70] and another based on interpolants [39]. We conjecture that there is an underlying similarity between the two methods, as both strive to compute a succinct explanation of the error trace by reducing the problem to reasoning about an unsatisfiable formula. Based on this, we are formulating a new hybrid algorithm that combines the benefits of both approaches. Our technique works by automatically reducing an error trace to its essential components—a minimal set of statements that are responsible for the error, together with key predicates that explain how these statements lead to the failure. We have proven that our approach is sound (i.e., captures all statements relevant to the error), and are working to show that it is useful for debugging real programs.

In conclusion, through this thesis, we believe to have set forth the stage for symbolic execution based reasoning and made the reader conversant with its intricacies and subtleties. We hope to have nurtured further research in this area, and conclude by stating that the future is bright for symbolic execution.

# Bibliography

- [1] Frama-C Software Analyzers. <http://frama-c.com/>.
- [2] TRACER. [paella.dl.comp.nus.edu.sg/tracer](http://paella.dl.comp.nus.edu.sg/tracer).
- [3] History's worst software bugs. <http://archive.wired.com/software/coolapps/news/2005/11/69355?currentPage=all>, August 2005.
- [4] Cambridge university study states software bugs cost economy \$312 billion per year. <http://undo-software.com/company/press/press-release-8>, January 2013.
- [5] OpenSSL TLS heartbeat extension read overflow discloses sensitive information. <http://www.kb.cert.org/vuls/id/720951>, April 2014.
- [6] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, 2012.
- [7] S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *TACAS*, pages 367–381, 2008.
- [8] G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, O. Wei, and A. Gupta. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *SAS*, 2008.
- [9] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*, pages 203–213.
- [10] L. Bent, D. C. Atkinson, and W. G. Griswold. A comparative study of two whole program slicers for C. Technical report, University of California at San Diego, La Jolla, CA, USA, 2001.

- [11] D. Beyer. Competition on software verification - (SV-COMP). In *TACAS*, 2012.
- [12] D. Beyer. Second competition on software verification. In *TACAS*, 2013.
- [13] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *PLDI'07*.
- [14] D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *CAV*, 2011.
- [15] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS*, pages 351–366, 2008.
- [16] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. In *ASE*, pages 443–446, 2008.
- [17] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, pages 209–224, 2008.
- [18] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *CCS*, pages 322–335, 2006.
- [19] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology*, 40, no. 11-12:595–607, 1998.
- [20] J. Christ, E. Ermis, M. Schaf, and T. Wies. Flow-sensitive fault localization. *VMCAI*, 2013.
- [21] D.-H. Chu, J. Jaffar, and V. Murali. Lazy Symbolic Execution for Enhanced Learning. In *RV*, 2014.
- [22] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. *CoRR*, 2013.



- [23] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *TACAS'08*, pages 397–412, 2008.
- [24] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterExample-Guided Abstraction Refinement. In *CAV'00*.
- [25] CodeSurfer. Grammatech Inc. <http://www.grammatech.com/products/codesurfer/>.
- [26] M. Colon, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.
- [27] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis. In *4th POPL*, pages 238–252. ACM Press, 1977.
- [28] W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.
- [29] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, 2012.
- [30] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
- [31] S. Danicic, C. Fox, and C. Harman. Consit: A conditioned program slicer. In *ICSM'00*, pages 216–226.
- [32] O. S. Daniel Kroening. Decision procedures: An algorithmic point of view, 2008.
- [33] M. Daoudi, L. Ouarbya, J. Howroyd, S. Danicic, M. Harman, C. Fox, and M. Ward. Consus: A scalable approach to conditioned slicing. *Working Conference on Reverse Engineering*, 2002.

- [34] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS*, 2008.
- [35] E. W. Dijkstra. Structured programming. chapter Chapter I: Notes on Structured Programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972.
- [36] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.
- [37] J. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [38] N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, 2011.
- [39] E. Ermis, M. Schäf, and T. Wies. Error invariants. In *FM*, 2012.
- [40] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 27:213–224, 2001.
- [41] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. *ICSE 2000*, 2000.
- [42] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 1987.
- [43] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *POPL '95*, pages 379–392.
- [44] J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *ESEC/FSE-13*, pages 227–236, 2005.
- [45] C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, 2010.

- [46] X. Gao, A. Snavely, and L. Carter. Path grammar guided trace compression and trace approximation. In *High Performance Distributed Computing*, 2006.
- [47] P. Godefroid. Compositional dynamic test generation. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 47–54. ACM Press, 2007.
- [48] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, pages 206–215, New York, NY, USA, 2008. ACM.
- [49] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.
- [50] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 2012.
- [51] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.
- [52] A. Gupta and A. Rybalchenko. InvGen: An Efficient Invariant Generator. In *CAV*, 2009.
- [53] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, pages 232–244. ACM Press, 2004.
- [54] B. Hetzel. *The Complete Guide to Software Testing*. QED Information Sciences, Inc., Wellesley, MA, USA, 2nd edition, 1988.
- [55] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 1969.
- [56] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.
- [57] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88*, pages 35–46.

- [58] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.
- [59] J. Jaffar, J. Navas, and A. Santosa. Unbounded Symbolic Execution for Program Verification. In *RV 2011*, pages 396–411, 2011.
- [60] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP( $\mathcal{R}$ ) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.
- [61] J. Jaffar and V. Murali. A Path-Sensitively Sliced Control Flow Graph. In *FSE*, 2014.
- [62] J. Jaffar and V. Murali. Trace Compression with Loop Explanations. Submitted, 2014.
- [63] J. Jaffar, V. Murali, and J. Navas. Boosting Concolic Testing via Interpolation. In *FSE*, 2013.
- [64] J. Jaffar, V. Murali, J. Navas, and A. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *CAV 2012*, pages 758–766, 2012.
- [65] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Path-sensitive backward slicing. In *SAS*, pages 231–247, 2012.
- [66] J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *CP*, 09.
- [67] J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *23rd AAI*, pages 297–303. AAI Press, 2008.
- [68] B. Jeannet and A. Mine. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, 2009.
- [69] R. Jhala and R. Majumdar. Path slicing. In *PLDI*, 2005.
- [70] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. *SIGPLAN Not.*, 46(6):437–446, 2011.

- [71] J. C. King. Symbolic Execution and Program Testing. *Com. ACM*, pages 385–394, 1976.
- [72] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
- [73] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *PLDI*, 2012.
- [74] G. Lalire, M. Argoud, and B. Jeannet. The Interproc Analyzer. <http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc>, 2009.
- [75] M. Li. A practical loop invariant generation approach based on random testing, constraint solving and verification. ICFEM'12, 2012.
- [76] X. Li, T. Mitra, H. S. Negi, and A. Roychoudhury. Design space exploration of caches using compressed traces. In *International Conference on Supercomputing*, 2004.
- [77] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1), 2008.
- [78] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS*, 2011.
- [79] Mälardalen WCET research group benchmarks. URL <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>, 2006.
- [80] K. L. McMillan. Lazy abstraction with interpolants. In *CAV '06*, pages 123–136.
- [81] K. L. McMillan. Interpolation and SAT-based model checking. In *15th CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.
- [82] K. L. McMillan. Lazy annotation for program testing and verification. In *22nd CAV*, 2010.
- [83] K. L. McMillan. Lazy annotation for program testing and verification. In T. Touili, B. Cook, and P. Jackson, editors, *22nd CAV*, volume 6174 of *LNCS*, pages 104–118. Springer, 2010.

- [84] V. Murali, N. Sinha, E. Torlak, and S. Chandra. A Hybrid Algorithm for Error Trace Explanation. In *VSTTE*, 2014.
- [85] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC'02*.
- [86] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, 1984.
- [87] A. Podelski and A. Rybalchenko. ARMC. In *PADL'07*.
- [88] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.
- [89] J. C. Reynolds. Separation logic: A logic for shared mutable data objects. In *17th LICS*, pages 55–74. IEEE Computer Society Press, 2002.
- [90] T. Robschink and G. Snelling. Efficient path conditions in dependence graphs. In *ICSE '02*, pages 478–488.
- [91] R. Rugina, M. Orlovich, and X. Zheng. Crystal: A program analysis system for C. <http://www.cs.cornell.edu/projects/crystal>, 2007. [Online; accessed 09-July-2011].
- [92] A. Rybalchenko. CLP-prover. URL <http://www7.in.tum.de/~rybal/clp-prover/>.
- [93] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. *POPL '04*, 2004.
- [94] A. Saswat. *Techniques to Facilitate Symbolic Execution of Real-world Programs*. PhD thesis, Georgia Institute of Technology, 2012.

- [95] P. H. Schmitt and B. Weiß. Inferring invariants by symbolic execution. In *Proceedings, 4th International Verification Workshop (VERIFY'07)*, 2007.
- [96] R. Sebastiani. Lazy satisfiability modulo theories. *JSAT*, 2007.
- [97] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, pages 263–272, 2005.
- [98] S. Seo, H. Yang, and K. Yi. Automatic construction of Hoare proofs from abstract interpretation results. In *APLAS'03*, pages 230–245.
- [99] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. volume 15, pages 410–457.
- [100] G. Snelting and A. Softwaretechnologie. Combining slicing and constraint solving for validation of measurement software. In *SAS*, pages 332–348, 1996.
- [101] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In *Logic-Based Program Synthesis and Transformation*, 1999.
- [102] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.
- [103] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
- [104] M. Weiser. Program slicing. In *ICSE '81*, pages 439–449, 1981.
- [105] T. Welp and A. Kuehlmann. QF BV model checking with property directed reachability. In *DATE*, 2013.
- [106] D. Wonisch. Block Abstraction Memoization for CPAchecker. In *TACAS*, 2012.

- [107] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pages 283–294, New York, NY, USA, 2011. ACM.
- [108] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE '02*, pages 1–10, 2002.
- [109] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *PLDI*, 2006.
- [110] X. Zhang and R. Gupta. Cost effective dynamic program slicing. *PLDI*, 2004.
- [111] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. *ICSE*, 2003.