Software Verification with Program-Graph Interpolation and Abstraction

by

Aws Albarghouthi

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Software Verification with Program-Graph Interpolation and Abstraction

Aws Albarghouthi
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2015

Picture a world where you can ask questions about a piece of code and have tools that automatically and efficiently answer them for you. *Can a division by zero ever occur? Are all elements in this list always greater than ten? Does this program always terminate? Are there any race conditions?* In such a world, software is considerably more reliable and secure than what we currently have; software has fewer bugs and is easily certifiable like other engineering artifacts; software-induced disasters are effortlessly avoided; and the billions of dollars that are normally spent on testing and maintenance activities are instead poured into more productive endeavours. Alas, such a world is an imaginary utopia, as the majority of these *verification* questions translate into *undecidable problems* (due to their relation to the halting problem). Nevertheless, we can still try to design algorithms that answer *some* questions about programs *most of the time*, and this is exactly what we do in this dissertation.

Specifically, we address the problem of automatically verifying safety properties of programs. Safety properties encompass a wide range of desirable correctness criteria (e.g., no assertions are violated, memory safety, secure information flow, etc.) and form the basis of other verification techniques for liveness properties (like termination). To prove that a program satisfies a safety property, we need to find a safe inductive invariant. A safe inductive invariant characterizes an over-approximation of reachable program states (a valuation of program variables) that does not intersect with unsafe states specified by the property. We advance safety property verification in several directions.

First, we target interpolation-based (IB) verification techniques. IB techniques are a new and efficient class of algorithms that avoids traditional abstract fixpoint invariant computation. IB techniques utilize the logical notion of Craig interpolants to hypothesize a safe inductive invariant by examining symbolic executions (finite paths) through the program. We propose *DAG interpolants*: a novel form of interpolants that facilitates efficient symbolic reasoning about exponentially many program paths represented succinctly as a DAG. In contrast, previous IB techniques explicitly enumerate paths and employ heuristics to avoid path explosion. We show how DAG interpolants can be used to construct efficient safety verification algorithms.

Second, we present an extension of McMillan's original IB algorithm [McM06] to the interprocedural

setting, enabling verification of programs with procedures and recursion. To do so, we introduce the notion of *state/transition interpolants*, and show how existing interpolant generation techniques can be used for hypothesizing procedure summaries.

Third, we propose new verification algorithms that combine abstraction-based (AB) verification techniques, based on abstract domains with fixpoint invariant computation, with IB techniques. Our new algorithms are parameterized by the degree with which AB and IB techniques drive the analysis, providing a spectrum of possible instantiations, and allowing us to harness the advantages of both IB and AB techniques. We experimentally demonstrate the effectiveness of our new hybrid IB/AB techniques and show that our algorithms can outperform state-of-the-art verification techniques from the literature.

Finally, we describe the design and implementation of UFO_{app}, an award-winning tool and framework for C program verification in which we implemented and evaluated our algorithms.

To my parents,
Majeda, Majeda, Majeda, and Hamza

Acknowledgements

I would like to thank Marsha Chechik, my amazing PhD advisor, for her patience, support, and encouragement over the course my graduate studies at UofT. Marsha believed in me as a budding researcher, gave me an unbelievable amount of academic freedom—perhaps more than I will ever have!—and always inspired me with her infinite enthusiasm. My meetings with Marsha were therapy sessions (for me). I would come in with an overwhelmingly messy pile of problems, issues, and questions. Marsha would then elegantly find order in my mess, clearing my mind and the path ahead. I hope that I can provide my future students with the same support that Marsha gave me.

All of the research presented in this dissertation was in collaboration with the magnificent and indefatigable Arie Gurfinkel. Arie generously gave me an enormous amount of his time; we spent hundreds upon hundreds of hours over email and Skype comping up with problems, restating problems, sketching algorithms, resketching algorithms, debugging our code, debugging other people's code, running experiments, rerunning experiments, writing papers, rewriting papers. I cannot imagine my graduate studies experience without Arie. He patiently endured my baby steps at the beginning of my career, always asked me the toughest questions, and ultimately shaped the way I do research. I have learned a lot from Arie, from how to punctuate an enumerated list, to how to properly manage pointers in C++, to how to make novel contributions and express them clearly. But the most crucial thing I learned from him is the importance of always pushing harder and further, particularly when I feel like I've done a good job, because that's what makes good work great. This lesson has not only made me a better researcher, but also a better person. Thank you, Arie.

I was inspired to go to graduate school by Wolfram Kahl, with whom I did undergraduate summer work on esoteric research problems. Even though I did not really understand what exactly I was working on at the time, I was thoroughly inspired by Wolfram's child-like love and unadulterated passion for research and programming. Thank you, Wolfram.

My first taste of research in graduate school came through taking a course with Sheila McIlraith. Sheila and Jorge Baier—the excellent course TA, who is now Prof. Baier—guided me through my first research project, showing me how to clearly formalize my ideas and how to write a scholarly paper. Although that project did not move past a workshop paper, it was a critical confidence boost at the outset of my studies that made me think, "Hey, I can actually do this!" Throughout the rest of my graduate studies, Sheila was constantly there for me as a mentor and as an advocate. Thank you, Sheila and Jorge.

The brilliant Azadeh Farzan has been my greatest career mentor. Azadeh served on my PhD committee and always went out of her way to help me out. Over many coffees, she discussed with me my career options, listened to my concerns, helped me immensely in my application process, and relentlessly critiqued and dissected my job talk. Azadeh taught me how to convincingly present my work: excite, intrigue, deliver (or was it intrigue then excite?). Without Azadeh's help, support, and advocacy, I wouldn't be where I am today. Thank you, Azadeh.

In the summer of 2011, I interned with the extraordinary duo of Aditya Nori and Sriram Rajamani at Microsoft Research in Bangalore, India. Ever since my internship, and despite being a few oceans away, Aditya has always been there for me, checking on me, steering me in the right trajectory, and placing opportunities along my path without me even knowing! Aditya is my guardian angel. Whatever you do, I wish you an Aditya in your life! Thank you, Aditya.

In the summer of 2012, I interned with the incredible Ken McMillan at Microsoft Research in Red-

mond. The work in this dissertation is directly inspired by, and builds upon, Ken's seminal work on interpolation. I was very fortunate to work with Ken and learn from him firsthand. Witnessing Ken conduct research is a pleasure. His ability to deeply immerse himself in a problem and emerge with a beautiful solution (pun intended) is stunning. I hope that someday I am able to reach Ken's level of Zen and see as deep into problems as he can. Thank you, Ken.

In the winter of 2013, I interned with the Terminator and the Slayer, Byron Cook and Josh Berdine, at Microsoft Research in Cambridge, UK. Josh kindly offered me a whole lot of time and pure mental power to discuss and refine my crazy ideas and teach me about the ins and outs of separation logic. During my stay in the UK, I was fortunate enough to get to know Peter O'Hearn. Byron and Peter believed in my research and have been of great support. Thank you, Byron, Josh, and Peter.

I would like to thank Ric Hehner for serving on my PhD committee—and staying on it after retiring! I would like to thank Rajeev Alur for agreeing to read this dissertation and serve as my external examiner on such a short notice. I would like to thank Sumit Gulwani for carving time out of his hectic sechedule to collaborate with two random graduate students from Toronto, and for being a great source of advice. I would also like to thank Isil Dillig for her indispensable advice and support in my job search over the past year.

My colleagues in the SE group made graduate school much more enjoyable, and I wholeheartedly thank them for that. Michalis Famelis always sees things from a refreshingly different perspective. With Michalis, I talked less about research and more about the world outside. We discussed a whole lot of topics, ranging from Marxism to space exploration! Alicia Grubb was always my resource for administrative questions. She knows the rules inside out, whether you're asking about requirements for PhD checkpoints or how to expense homeopathy clinic visits on student insurance! I've always wondered why Alicia is in computer science and not in law. Zachary Kincaid was, and will continue to be, my resource for everything research. Zak is the smartest person I've worked with. He has the exceptional ability to see through problems and crack them effortlessly. I would sketch a problem I'm having for Zak on the whiteboard, and after staring at it and pacing around for two silent minutes, he would walk up to the whiteboard and write the solution. Be careful with showing Zak your problems, he'll solve them in a couple of minutes and take all the credit! When I was junior graduate student, Jocelyn Simmonds, now Prof. Simmonds, gave me tonnes of advice on research and enthusiastically answered all of my naive questions. She also caught the lab thief! (Long story for the acknowledgments section.) Ou Wei, now Prof. Wei, got me into program analysis; without him, this thesis probably wouldn't have happened. When I started, he showed me Aditya and Sriram's paper on the Synergy algorithm, and I said to myself, "I want to go to India and work with Aditya and Sriram!" Yi Li joined the UFO project at its peak, did excellent work, and, like me, got bit by the program analysis bug! I would also like to thank my SE group colleagues Jennifer Horkoff, Golnaz Elahi, Jorge Aranda, Neil Ernst, Lev Naiman, Alessio Di Sandro, Rick Salay, Julia Rubin, Ahmed Mashiyat, and Albert Lai.

In my intercontinental graduate school adventures, I met many great friends who made my graduate studies experience truly wonderful: Sam Blackshear, Ted McCarthy, Rishabh Singh, Loris D'Antoni, Jose Falleiro, Nithya Sambasivan, Gaurav Paruthi, Heidy Khlaaf, and many others.

To the NSERC committee that decided to award me the CGS PhD scholarship: Thank you; this helped make my graduate school experience a much more comfortable one. (To the NSERC committee that denied me the postdoctoral fellowship and ranked my application in the bottom eight percent: I got a faculty position instead. How do you like me now?!)

Maria Alejandra, my wife and life partner, endured all the deadlines, the travel, the internships. During all of this, Maria gave me unlimited and unwavering support. And when I say support, I don't mean support in the clichéd sense, I mean support as in physical fortification: with Maria's support I could go through walls and come out the other side unscathed. When I was down Maria lifted me up, and when I was up she lifted me higher. Without her, I wouldn't have been able to reach the end of this journey. Thank you and love you forever, Maria.

Although six years my junior and views me as his older brother, I learn from Adam more than he learns from me. His strength, passion, and view of life inspire me everyday, and I cannot wait to see the great things he will create. Adam always asks me tough questions that alter the way I see the world. He is constantly making me a better person. Thank you, Adam, for your love, care, support, and encouragement.

The amount of effort my parents, Majeda and Hamza, have invested, from before day zero, to get me where I am today is immeasurable. They've selflessly dedicated their lives to giving me and Adam better lives. They've been there for me with all they've got in every moment of the past twenty-seven years. It would take many, many lifetimes of full-time work to repay the gargantuan debt that I owe them. Thank you, *Yamma w Yaba*. This humble dissertation is dedicated to you.

Aws Albarghouthi
Toronto, Canada
November 2014

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Preamble

Over the past few decades, we have witnessed software systems invading every facet of our life. With our increased reliance on software, both at the personal and organizational level, the consequences of software failure can transcend mere annoyance and have profound negative effects on our lives. Thus, tools and techniques for rigorous analysis and reasoning about software are ever more important.

The simplest and most used technique for reasoning about a piece of software is to test it. While great advances have been made in testing technologies, both in academia and industry, testing is usually insufficient for guaranteeing safe program operation. To appeal to Edsger Dijkstra's famous quote [Dij72], *"Program testing can be used to show the presence of bugs, but never to show their absence!"* In other words, testing only explores a small subset of the possible behaviours of a program; therefore, it does not supply guarantees on all possible behaviours. This is where the problem of *software verification* comes into play: *proving, mathematically, that a program satisfies some desired property*, e.g., memory safety, termination on all inputs, or some program-specific functional specification like the program always returns a positive integer.

Software verification is a classic problem that dates back to Alan Turing's proof of undecidability of the halting problem [Tur36], which effectively eliminated all hope for an automatic procedure for proving program termination, and as a corollary, most desirable properties of programs.

Turing's proof of undecidability of the halting problem did not deter scientists from studying manual and automated techniques for verifying software. Indeed, the importance of being able to formally reason about software, particularly in our increasingly computerized world, promulgated verification to the forefront of a number of major computer science research communities (which have enjoyed quite a few Turing Awards over the years). The work on software verification started in the sixties and seventies[1] with mathematical frameworks for manually reasoning about programs, paving the way for automated techniques in the eighties, nineties, and aughts.

This dissertation continues this long and rich tradition of software verification research by contributing novel algorithmic techniques for *automatically verifying safety properties of programs*, with the overarching goal of advancing the efficiency and applicability of automated verification techniques.

A program state is a valuation of all variables of the program (including the program counter). A

[1] Needless to say, we are talking here about the 20^{th} century!

safety property specifies a set of bad (unsafe) states that the program should never be in. A program is correct with respect to a safety property if there is no execution that can reach a bad state specified by the property. We are concerned with the problem of automatically proving a program correct with respect to a safety property.

In this first chapter, we paint a wide (but incomplete) picture of software verification research over the past few decades and provide a detailed view of modern automated safety verification techniques (Section 1.2). We then state the main contributions of this dissertation and describe how it advances the state of the art of automated verification (Section 1.3).

1.2 A (Partial) History of Software Verification

1.2.1 The Early Days

One of the first to recognize the practical importance of reasoning about software was Turing himself—the person who proved undecidability of the problem. In a paper titled *"On Checking a Large Routine"* from 1949 [Tur49], Turing starts by asking, *"How can one check a routine in the sense of making sure that it is right?"* He then proceeds to sketch a proof of two properties of a program that computes the factorial, $n!$, of its input $n \in \mathbb{Z}$: (1) the program always terminates with a result on all inputs and (2) always returns the factorial of its input parameter. The former is now known as a *liveness property*: the program will eventually do something good (in this case, terminate with some result). The latter is now considered a safety property: the program should never do something bad (in this case, return an incorrect value). As mentioned, this dissertation focuses on verifying safety properties of programs, which specify that the program should not reach an undesired (bad) state—in Turing's factorial program, this is a state where the program reaches the return statement with a return value that is not equal to $n!$. We note that modern liveness verification techniques, for example, Cook et al.'s Terminator tool [CPR06], reduce liveness checking to checking a sequence of safety properties [CPR05]. Thus, progress in safety property verification directly and positively impacts liveness property verification.

A little more than a decade after Turing's 1949 paper—as computers started to play a bigger role in industrial and academic life—early computer science pioneers recognized the importance of, as Floyd [Flo67] concisely put it, *"assigning meanings to programs,"* that is, viewing programs as mathematical artifacts that we can formally reason about. As a result, Floyd-Hoare logic [Flo67, Hoa69] and Dijkstra's predicate transformers [Dij75] introduced a logical framework for deducing the reachable states of a program, thus providing a disciplined approach for manual program verification and laying the theoretical underpinnings of the (semi-)automated verification techniques to come.

1.2.2 The Age of Automation

The road towards automatic software verification started with two almost independent lines of research born in the late seventies and early eighties:

- *Model checking*, initiated independently by Clarke and Emerson [CE81] and Quielle and Sifakis [QS81], started as an algorithmic technique for checking if a given structure is a model of a formula in temporal logic. Models of temporal logics, like *Linear Temporal Logic* (LTL) [Pnu77], are *Kripke structures* (finite state machines). Thus, by viewing a software or hardware system as a finite

state machine, model checking offers an automated way of verifying sophisticated temporal logic properties, which encompass a wide range of safety and liveness properties.

Model checking relies on algorithmically enumerating all the states of a Kripke structure in order to determine if it satisfies a temporal logic specification. Unfortunately, when dealing with real programs, the state space can be prohibitively large or even infinite (for example, due to arbitrary precision integers or unknown size of memory). Due to this limitation, the success of model checking was constrained to hardware and protocol verification, which typically give rise to smaller state spaces. But even for hardware and protocol verification, efficient model checking required significant algorithmic advances that came in the form of symbolic techniques for succinctly representing large sets of states as formulas. Specifically, *Binary Decision Diagrams (BDDs)* [Bry86] and efficient *satisfiability (SAT) solvers* [MZ09] gave rise to *symbolic model checking* [McM93] and *bounded model checking* [BCCZ99] techniques.

- Cousot and Cousot's *abstract interpretation* framework [CC77] provided a unifying lens with which we can view program analysis and verification techniques: as over-approximations (abstractions) of the concrete semantics of a program. Specifically, Cousot and Cousot showed how data-flow analyses used in compiler optimizations (for example, constant propagation and live variable analysis) and Floyd-Hoare proofs can be viewed as an "execution" of an abstract version of the program where only a few facts are tracked and the rest are thrown away. For instance, a live variable analysis executes the program while only tracking what variables are live at each program location, dismissing what values these variables actually hold. The abstract interpretation framework provided a disciplined way of (1) defining abstractions, known as *abstract domains*, of concrete program semantics and (2) building program analyses over these abstract domains, thus allowing us to compute over-approximations of reachable program states.

The last fifteen years saw an explosion in automated software verification techniques that can be applied to real programs with thousands of lines of code. Advances in model checking, abstract interpretation, and automated theorem proving conspired to create this breakthrough. We highlight the main advances below:

- *Predicate abstraction* [GS97] provided a family of abstract domains that over-approximate the semantics of a program and result in a finite-state abstraction of the program (where each state in the finite abstraction represents possibly infinitely many concrete program states). This enabled direct application of classic model checking approaches to programs which might have large or infinite state spaces.

- To enable construction of finite-state program abstractions, heavy use of automated theorem proving is required. Luckily, the early aughts also witnessed significant breakthroughs in SAT and *Satisfiability Modulo Theories* (SMT) solving [BSST09]. SMT solvers capitalized on the algorithmic and engineering advances of SAT solvers in order to reason about a rich subset of first-order logic. This includes first-order theories such as bitvectors, linear arithmetic, and arrays. These advances facilitated precise modeling of, and reasoning about, program semantics.

- An abstraction of a program might be too coarse, resulting in false positives. In other words, an abstraction might throw away too much information, causing verification to conclude that a bad

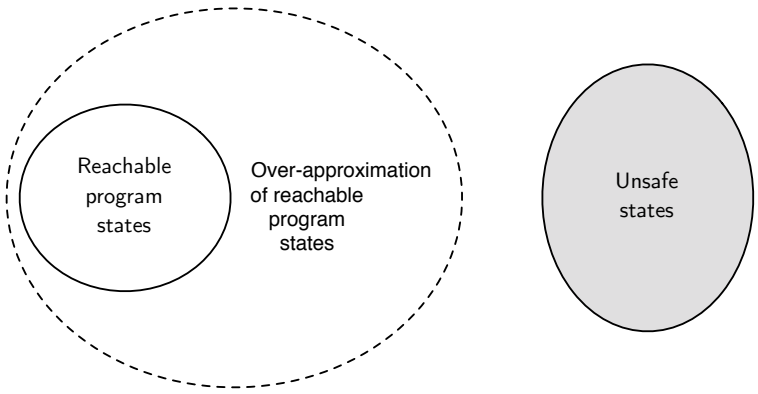


Figure 1.1: Illustration of over-approximations of reachable states.

state is reachable when it is not. The *Counterexample-Guided Abstraction Refinement* (CEGAR) framework [CGJ+00] offered a solution to this problem. Specifically, given a counterexample (a faulty execution) found by analyzing the finite-state abstraction, CEGAR either confirms that the counterexample is real (maps to a faulty execution under the concrete semantics) or proposes a refined (less coarse) abstraction in which this counterexample is eliminated. Given the general undecidability of the verification problem, an abstraction might keep getting refined indefinitely!

Perhaps the most notable application of predicate abstraction and CEGAR is within the Slam project [BR01]. The Slam project built an industrial-grade toolchain for verifying API-usage properties of Windows device drivers, and inspired huge interest in automated software verification research.

- The aforementioned advances relied on a two-step process: (1) computing a finite-state program abstraction with the help of predicate abstraction and automated theorem proving, and (2) utilizing finite-state model checking techniques for proving program safety. As a result, in the literature, they fall under the umbrella of so-called *software model checking* techniques. In parallel to software model checking techniques, numerical abstract domains were also used to verify properties of real programs; most notably, proving run-time safety of aircraft software [BCC+03]. Unlike the abstract domains typically used in software model checking, most numerical abstract domains (for example, *intervals* [CC76] and *octagons* [Min06]) do not yield finite-state abstractions, and instead depend on over-approximation (widening) strategies in order to force the analysis to terminate. These domains are considered infinite-height domains: they represent lattices of infinite height, where elements higher in the lattice capture more concrete program states. Conversely, predicate abstraction domains are finite-height domains (since they yield finite-state abstractions).

We note that our brief survey is biased towards the focus of this dissertation and thus neglects important classes of work on program correctness. We do not discuss the huge fields of type systems and interactive proof assistants. We also do not discuss semi-automated (deductive) verification techniques. It is important to also note that manual and semi-automated verification remain very active areas of research, particularly for complex properties and programs that cannot be handled by existing automated techniques.

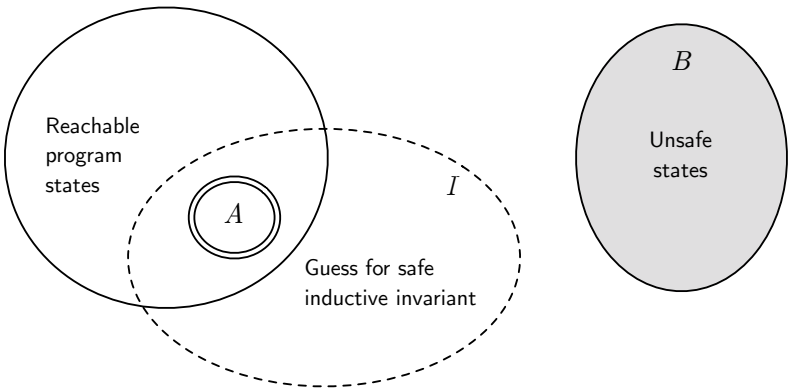


Figure 1.2: Illustration of interpolation-based verification.

1.2.3 Two Automated Verification Techniques

In principle, all safety verification techniques compute an over-approximation of the set of reachable program states, called an *inductive invariant*. An invariant is an over-approximation I of the set of reachable program states. An inductive invariant is one where executing the program from any state in I results in a state that is also in I. A safety property defines a set of unsafe program states. Thus, if the inductive invariant does not intersect with the set of unsafe states, then it constitutes a proof that the program is correct with respect to the given safety property—we say it is a *safe* inductive invariant. Figure 1.1 illustrates this idea.

The key question is: How do we compute an inductive invariant? We categorize contemporary automated verification techniques into two closely related classes, differentiated by the method used to construct a safe inductive invariant.

- **Abstraction-based (AB) techniques**: AB techniques utilize an abstract domain (e.g., predicate abstraction) that over-approximates the semantics of program statements. The program is executed under the abstract semantics, while collecting all abstract states encountered along the way. The process stops when no new abstract states can be found, i.e., an inductive invariant has been computed. Most automated verification techniques fall under this class, e.g., software model checking with predicate abstraction, abstract interpretation with numerical domains, etc.

  Intuitively, the abstract domain restricts the language with which we can define an inductive invariant. For instance, the intervals numerical domain restricts invariants to formulas of the form $\bigwedge_i l_i \leqslant x_i \leqslant u_i$, where $\{x_i\}_i$ are program variables and $\{l_i, u_i\}_i$ are numerical constants. Imposing a restriction on the logical language makes it easier to systematically search for an inductive invariant. For example, a predicate abstraction domain defines a finite set of candidate invariants; thus, we can simply search through all candidates until we arrive at a safe inductive invariant (of course, this might not be the most efficient strategy).

  The main disadvantages of abstraction-based verification are two-fold: First, an abstract domain might be too weak to construct a safe inductive invariant, e.g., no safe inductive invariant is expressible in the restricted language imposed by the abstract domain. Second, executing the program under abstract semantics is often very expensive, for example, involving worst-case exponential

operations in the case of predicate abstraction. This is known as an *absctract post operation*: executing a program statement starting from an abstract state to arrive at a new abstract state.

- **Interpolation-based (IB) techniques**: As an alternative to AB techniques, McMillan introduced IB techniques first for hardware [McM03] and then for software verification [McM06], and showed that they can outperform AB techniques. The key advantage of IB techniques over AB techniques is that they do not restrict the search for an inductive invariant with an abstract domain; thus, they avoid the expensive abstract post computation required by AB techniques. At a high level, IB techniques work as follows:

  1. Pick some finite execution paths through the control-flow graph of the program and encode them as first-order formulas (in a manner similar to bounded model checking or *symbolic execution* [Kin76]).
  2. The formulas represent a subset (an under-approximation) of the reachable program states. In Figure 1.2, this is represented by the subset, A, with double borders. If the subset intersects with set of unsafe states, B, then we know that the program is unsafe. Otherwise, *Craig interpolants* [Cra57] are computed to over-approximate this subset while making sure that the over-approximation does not intersect with the unsafe states. One such over-approximation, I, is shown in Figure 1.2 with a dashed border. Effectively, this over-approximation serves as a *hypothesis* (an educated guess) for a safe inductive invariant. Obviously, the hypothesis in our figure is not invariant, since it does not encompass all reachable states. In this case, the process continues by examining a larger subset of reachable states that includes A and refining the hypothesis.

     Given two formulas A and B in first-order logic, where $A \wedge B$ is inconsistent, a Craig interpolant is a formula I over the shared symbols of A and B, where $A \Rightarrow I$ and $I \Rightarrow \neg B$. Thus, if we view A as our subset of reachable states and B as our set of unsafe states, an interpolant is an over-approximation of our subset of reachable states that does not intersect with the unsafe states. McMillan [McM03] showed that interpolants can be efficiently extracted from *refutation proofs* produced by SAT solvers; a flood of later works extended the idea to other SMT theories. Note that the hypotheses computed using interpolants can be arbitrary formulas within the logic used to encode program paths and unsafe states; therefore, hypotheses (and inductive invariants) are not restricted to an abstract domain.

  IB techniques examine concrete program states; this enables them to potentially find counterexamples faster than AB techniques that rely on CEGAR to confirm or refute abstract counterexamples. The main disadvantage of IB techniques is that they are merely making hypotheses that may or may not result in an inductive invariant—informally, one can view them as unguided. On the other hand, AB techniques are eagerly constructing an inductive invariant. Thus, in cases where abstract post computation is cheap and the abstract domain is sufficient, AB techniques might arrive at an answer faster than IB techniques.

One may argue that there is no distinction between IB and AB techniques. For instance, one may argue that the fragment of first-order logic used for interpolation is an abstract domain used by an IB technique. Indeed, we do agree with that: any logic used to model concrete program semantics can be

viewed as an abstract interpretation of concrete program semantics. Our distinction here is operational and philosophical:

- At the algorithmic level, IB techniques do not employ a forward abstract fixpoint computation like AB techniques, and thus do not execute an abstract version of the program. In other words, AB techniques spend most of their time in abstract post computation (and only occasionally perform other operations such as refinement). On the other hand, IB techniques spend most of their time examining program paths by encoding their concrete semantics and proving their safety using automated theorem proving.
- At the philosophical level, the logic used to model program semantics can be viewed as an abstract domain, but it is much less restrictive than traditional abstract domains, where strong syntactic requirements are imposed on the invariants with sole goal of enabling abstract fixpoint computation.

1.3 Challenges and Contributions

In the previous section, we gave an overview of modern automated verification techniques, categorizing them into abstraction-based and interpolation-based. ***The high level contribution of this dissertation is new verification algorithms that push the frontiers of interpolation-based verification, making it efficient and practical, while incorporating ideas from abstraction-based techniques.*** The following discussion explicates our individual contributions. Figure 1.3 helps outline our contributions with respect to IB and AB techniques.

1: Verification with DAG Interpolants (Chapter 3) Craig interpolants [Cra57] made their way into verification literature and tools through McMillan's seminal work on hardware model checking [McM03]. Building on the success of bounded model checking (BMC) with SAT solvers [BCCZ99], McMillan showed how to exploit the resolution proof produced by a SAT solver for a BMC problem to over-approximate the reachable states of a finite unrolling of a transition relation (bounded executions of the program). The key insight is that in the course of a resolution proof, a SAT solver makes decisions on which variables are important or relevant (the ones on which it resolves). By traversing the resolution proof bottom-up and focusing on relevant states, McMillan showed how to construct a formula, an interpolant, that over-approximates reachable states through a bounded unrolling of a problem and acts as a guess for a safe inductive invariant, thus, extending bounded model checking to the unbounded case.

Interpolants eventually made their way into infinite-state software verification. First, in the work of Henzinger et al. [HJMM04], interpolants were used for abstraction refinement in the CEGAR framework. Specifically, given an infeasible program path to an error location, interpolants were used to compute new predicates to refine (strengthen) a predicate abstract domain in order to eliminate the infeasible program path and possibly others. This approach was implemented with success in the Blast software model checker's lazy abstraction algorithm [HJMS02].

In his later work on *lazy abstraction with interpolants* (LAWI), McMillan used interpolants to directly compute inductive invariants. That is, instead of using interpolants as a means for refining a predicate abstract domain, McMillan showed how the interpolants themselves can be used to construct the inductive invariant, in a style similar to their initial use in finite-state model checking [McM03]. This

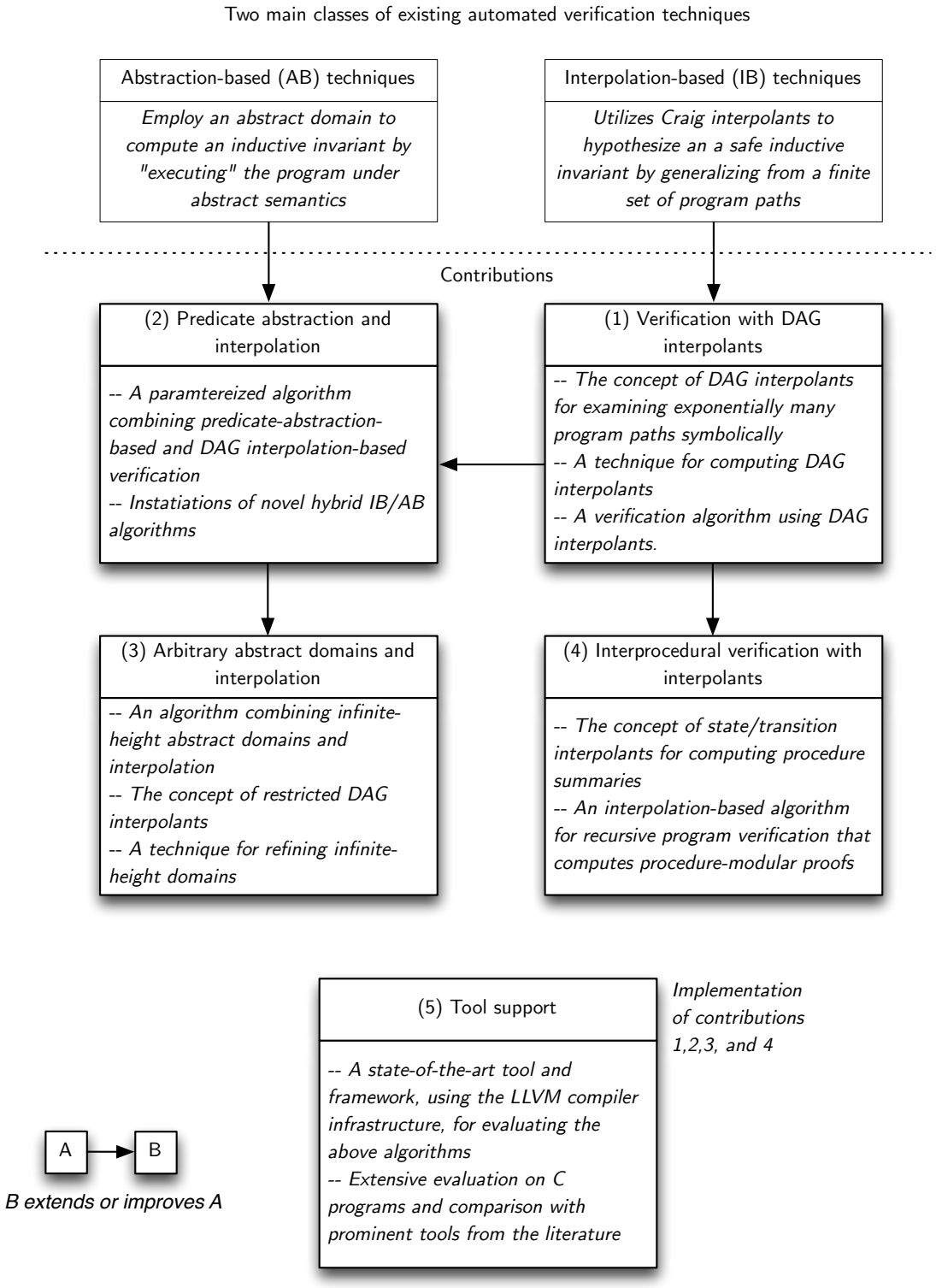


Figure 1.3: The five major contributions of this dissertation and the dependencies between them.

approach is what we call an interpolation-based (IB) software verification technique in this dissertation, as it eschews use of abstract domains and abstract fixpoint computation. To achieve this for software verification, new interpolation procedures were proposed for first-order theories like linear arithmetic, arrays, and bitvectors [HJMM04, JM07, KW07].

At a high level, LAWI works by sampling finite paths through the control-flow graph of the program to an error location (e.g., location of an assertion violation), and then uses interpolants to compute a *Hoare-style* [Hoa69] proof of each path. The semantics of instructions along each sampled path are encoded as a sequence of formulas, a formula per instruction. If the conjunction of the sequence of formulas is unsatisfiable (equivalent to *false*), then *sequence interpolants* are computed from the proof of unsatisfiability (refutation proof). Sequence interpolants form a Hoare-style proof of infeasibility of the path, that is, a proof that no concrete execution through the path can reach the error location.

Even a program with no loops can have exponentially many paths in the size of its control-flow graph—the simplest example is a program with a sequence of if-then-else statements. In such cases, LAWI might end up sampling a huge number of paths before arriving at an inductive invariant. A number of heuristics are proposed in [McM06] for dealing with path explosion.

One of the main insights in this dissertation is that we can compute proofs for a large number of symbolically-encoded paths in a single shot, instead of mechanically enumerating them. We demonstrate how to exploit the enumerative power of SMT solvers to achieve this. Specifically, in this chapter, we introduce the concept of *Directed Acyclic Graph (DAG) interpolants*. DAG interpolants extend the concept of an interpolant between two formulas, or a sequence of formulas, to a set of formulas spatially arranged in a DAG structure. Given a technique for computing DAG interpolants, we can compute proofs for a set of program paths succinctly encoded as a DAG, where every path through the DAG represents a program path.

Armed with a procedure for computing DAG interpolants, we show how to construct a verification algorithm by systematically unrolling the control-flow graph into a DAG (instead of a tree) and using DAG interpolants to hypothesize a safe inductive invariant.

2: Integrating Predicate Abstraction and Interpolation (Chapter 4) As mentioned in Section 1.2, AB and IB techniques offer different sets of complementary advantages. IB techniques completely avoid post operators, but might get stuck making incorrect hypotheses for a long time. AB techniques, on the other hand, eagerly try to compute an inductive invariant, but can spend too much time if the post operator is expensive or if the abstract domain used is insufficient and requires considerable refinement using CEGAR.

We propose a novel algorithm that integrates predicate-abstraction-based verification (an AB technique) with our DAG-interpolation-based verification algorithm (an IB technique). The algorithm is parameterized by the degree with which AB and IB approaches drive the analysis, providing a spectrum of possible instantiations and allowing us to harness the advantages of both IB and AB techniques. At one extreme, it is an AB algorithm where a predicate abstraction domain computes inductive invariants and interpolants are only used to refine the abstract domain. At the other end of the spectrum, it is an IB algorithm where no abstract domain is used and DAG interpolants hypothesize inductive invariants. In the middle of the spectrum, the algorithm can be instantiated as a hybrid IB/AB technique, where interpolants hypothesize an invariant I, and predicate abstraction tries to "fix" it by making it a safe inductive invariant I'.

We perform an extensive experimental evaluation and show that our hybrid IB/AB instantiations of the algorithm can outperform pure IB and AB techniques. Further, we show our DAG interpolation–based IB technique outperforms an implementation of McMillan's original IB algorithm [McM03].

3: Integrating Abstract Domains and Interpolation (Chapter 5) Contribution 2 introduces an integrated predicate abstraction and interpolation algorithm. We take this idea one step further by admitting arbitrary abstract domains for the AB portion of the algorithm, instead of just predicate abstraction domains. That is, we allow infinite-height domains, like intervals, boxes, or octagons, to be used for abstract fixpoint computation. Extending the algorithm in this direction allows us to experiment with different abstract domains (including highly efficient ones like intervals) that might be suited for different classes of programs. Additionally, we introduce the concept of *restricted DAG interpolants*: an extension of DAG interpolants that allows us to utilize the results of AB in order to improve the "quality" of hypotheses made by interpolants.

In generalizing our algorithm to arbitrary abstract domains, we faced a number of technical challenges. For instance, (1) maintaining disjunctive invariants in the form of a DAG (in order to apply DAG interpolation) and (2) extending widening operations of a given abstract domain to its finite powerset. The resulting algorithm can be viewed through two lenses:

1. We can view abstract domains as a form of guidance for our DAG interpolation procedure. That is, we can view our algorithm as an IB algorithm where information computed using the AB portion is simply used to enhance the interpolation process.

2. Alternatively, we can view our algorithm as a technique for refining unsafe inductive invariants (computed with abstract domains) using interpolation. Specifically, we place invariant generation with abstract interpretation within a refinement loop, where DAG interpolants are used to strengthen inductive invariants computed with abstract domains. That is, interpolants are simply used to eliminate *false alarms* (false reports of bugs) incurred due to coarse abstractions and over-approximating operations typical of abstract domains (e.g., *join* and *widening*). This is one of the first works to place general abstract-domain-based analyses in a refinement loop.

We experiment with different abstract domains and show that the resulting technique can outperform state-of-the-art tools as well as our previous algorithms.

4: Interprocedural Verification with Interpolation (Chapter 6) Contributions 2 and 3 are restricted to *intraprocedural* analysis, i.e., analysis of programs with a single function. We propose a novel *interprocedural* verification algorithm that is interpolation based. Specifically, we introduce the notion of *state/transition interpolants* and demonstrate how they can be used to hypothesize procedure summaries in order to compute modular proofs of correctness for programs with procedure calls and recursion.

Our contribution is an extension of McMillan's IB algorithm [McM03] to the interprocedural setting. Whereas [McM03] unrolls the control-flow graph of a program, our algorithm unrolls the call graph, and uses state/transition interpolants to label it with procedure summaries. In a fashion similar to DAG interpolants, state/transition interpolants enable symbolic reasoning about a set of DAGs, where each DAG in the set represents the control-flow graph of a procedure. To the best of our knowledge, this is the first extension of [McM03] that computes modular interprocedural proofs.

5: Building Efficient Verification Tools (Chapter 7) To facilitate and experiment with the algorithmic contributions of this dissertation, we built a state-of-the-art program analysis and verification framework, called UFO_{app}, using the LLVM compiler infrastructure [LA04]. We used UFO_{app} to verify safety properties of a large set of C programs. UFO_{app} competed in the 2013 International Software Verification Competition [Bey13] and won four out of ten verification categories—all the categories of programs for which it was designed. We provide a detailed description of UFO_{app}'s extensible architecture and prominent optimizations.

A Note on our Contributions Contributions (2) and (3) are both intraprocedural verification algorithms (Ufo and Vinta, respectively). Contribution (3) strictly generalizes (subsumes) and improves Contribution (2) by enabling general abstract domains to be used as the AB portion of the algorithm; Contribution (2) is restricted to predicate abstract domains.

Contribution (4) is an interprocedural verification technique, allowing us to directly handle recursive programs. Note that, unlike (2) and (3), Contribution (4) is purely interpolation-based, i.e., it does not utilize abstraction-based techniques to aid the interpolation process. In Section 6.8, we discuss the relative (dis)advantages of using interprocedural versus intraprocedural verification, given that we can convert a program with a single procedure to a recursive program and vice versa.

1.4 Organization of this Dissertation

The rest of this dissertation is organized as follows:

- In Chapter 2, we formalize the main concepts required for describing our contributions.
- In Chapter 3, we introduce and formalize DAG interpolants, present a procedure for computing them, and show how to build a software verification procedure using DAG interpolants. (*Based on* [AGC12b].)
- In Chapter 4, we introduce a parameterized verification algorithm that combines predicate abstraction and DAG interpolation. We then present an experimental evaluation of different instantiations of our algorithm. (*Based on* [AGC12b].)
- In Chapter 5, we extend the algorithm from Chapter 4 to arbitrary abstract domains, present the enabling concept of restricted DAG interpolants, and demonstrate its practical merit experimentally. (*Based on* [AGC12a].)
- In Chapter 6, we present an interprocedural verification algorithm that computes procedure-modular proofs using the novel notion of state/transition interpolants. (*Based on* [AGC12d].)
- In Chapter 7, we describe the design and implementation of UFO_{app}, our verification tool and framework in which we implemented our algorithms for experimentation and evaluation. (*Based on* [AGC12c, AGL+13].)
- Finally, in Chapter 8, we summarize our contributions and discuss open problems and future research directions.

Appendix A contains a simple example illustrating inductive invariants.
Appendix B contains proofs of lemmas and theorems whose proofs do not appear in the main text.

Chapter 2

Background

In this dissertation, we are concerned with the problem of proving program safety. In this chapter, we present a number of core concepts required for describing our contributions in the rest of this dissertation.

2.1 Programs and Safety

Programs A *program* P is a tuple $(\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$, where

- $\mathcal{L}$ is a finite set of *control locations*,
- δ is a finite set of *actions*,
- $\mathsf{en} \in \mathcal{L}$ is a special control location denoting the *entry* location of P,
- $\mathsf{err} \in \mathcal{L}$ is the only *error* location, and
- Var is the set of variables of program P.

An *action* $(\ell_1, T, \ell_2) \in \delta$ represents a program statement between ℓ_1 and ℓ_2, where $\ell_1, \ell_2 \in \mathcal{L}$ and T is the program statement. We assume that there does not exist an action $(\ell_1, T, \ell_2) \in \delta$ such that $\ell_1 = \mathsf{err}$.

A statement T can be an assignment $\mathtt{x} := E$ or assume statement $\mathtt{assume}(B)$, where $\mathtt{x}$ is a variable in Var, E is an expression over Var, and B is a Boolean expression over the variables in Var.

A statement T can be viewed as a *transition relation* over the variables $\mathsf{Var} \cup \mathsf{Var}'$, where Var' is the set of primed versions of variables in Var. We write $[\![T]\!]$ for the standard semantics of a statement T. For example, if T is `x := x + 1`, then $[\![T]\!] \equiv x' = x + 1$. If T is `assume(x > 0)`, then $[\![T]\!] \equiv x > 0$. Throughout this dissertation, we use `type writer` fonts for program variables and statements, and *math* fonts for their mathematical denotation.

Program Safety Given a program $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$, we say that P is safe if and only if every execution that starts in en, with any valuation of the variables Var, never reaches err. More formally, P is safe if there exists an annotation $Inv : \mathcal{L} \to B$, a safe inductive invariant, from program locations to Boolean formulas over Var such that

1. $Inv(\mathsf{en}) \equiv \mathit{true}$,
2. $Inv(\mathsf{err}) \equiv \mathit{false}$, and
3. for all $(\ell_1, T, \ell_2) \in \mathcal{L}$, $Inv(\ell_1) \wedge [\![T]\!] \Rightarrow Inv(\ell_2)'$.

The annotations Inv form a Hoare-style proof of correctness, where each location in the program is annotated with a formula encoding an over-approximation of reachable states at that location. For any action $(\ell_1, T, \ell_2) \in \mathcal{L}$,

$$\{Inv(\ell_1)\}\ T\ \{Inv(\ell_2)\}$$

is a valid *Hoare triple* [Hoa69], an axiom stating that if we execute the statement T starting from any state in $Inv(\ell_1)$, we will arrive at state in $Inv(\ell_2)$. Annotations of loop heads (*cutpoints*) are loop inductive invariants. For an example illustrating safe inductive invariants and how they are computed, we refer the reader to Appendix A.

Strongest Postcondition Let T be a statement in some program P with variables Var. Let φ be a formula over Var (representing a set of program states). We define the *strongest postcondition*, $\mathsf{SP}(\varphi, T)$, as the strongest formula with free variables in Var such that $\varphi \wedge [\![T]\!] \Rightarrow \mathsf{SP}(\varphi, T)'$ is valid. Informally, $\mathsf{SP}(\varphi, T)$ represents the set of states reachable by executing T from any state that satisfies φ. Concretely, $\mathsf{SP}(\varphi, T)'$ is equivalent to

$$\exists \mathsf{Var}.\varphi \wedge [\![T]\!].$$

2.2 Abstract Reachability Graphs

We now present *Abstract Reachability Graphs* (ARGs). ARGs are data structures we use to compute a safe inductive invariant for a given program. ARGs are a generalization of *Abstract Reachability Trees* (ARTs) [McM06, HJMS02] to directed acyclic graphs.

Definition 2.1 (Abstract Reachability Graph (ARG))**.** Let $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ be a program. An ARG $\mathcal{A}$ of P is a tuple $(V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$, where

- (V, E, v_{en}) is a directed acyclic graph (DAG) rooted at the *entry node* $v_{\mathsf{en}} \in V$;
- $\nu : V \to \mathcal{L}$ is a map from nodes to locations of P, where $\nu(v_{\mathsf{en}}) = \mathsf{en}$;
- $\tau : E \to \delta$ is a map from edges to actions of P such that for every edge $(u, v) \in E$, there exists an action $(\nu(u), \tau(u, v), \nu(v)) \in \delta$; and
- $\psi : V \to B$ is a map from nodes V to Boolean formulas over Var. □

A node v such that $\nu(v) = \mathsf{err}$ is called an *error node*. A node $v \in V$ is *covered* if and only if there exists a node $u \in V$ that dominates v and there exists a set of nodes $X \subseteq V$ such that

- $\forall x \in X \cdot x$ is uncovered,
- $\psi(u) \Rightarrow \bigvee_{x \in X} \psi(x)$, and
- $\forall x \in X \cdot \nu(u) = \nu(x) \wedge u \not\preceq x$,

where $\preceq$ is the ancestor relation on nodes and all $x \in X$ are less than u according to some fixed total order on nodes V. We say that node u is covered by X (when $X = \{v\}$, i.e., a singleton set, we will say u is covered by v). A node u *dominates* v if and only if all paths from v_{en} to v pass through u. By convention, every node dominates itself.

Definition 2.2 (Well-labeledness of ARGs)**.** Given an ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$ of a program $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ and a map $\mathcal{M}$ from every $v \in V$ to a Boolean formula over Var, we say that $\mathcal{M}$ is a *well-labeling* of $\mathcal{A}$ if and only if

1. $\mathcal{M}(v_{\mathsf{en}}) \equiv \mathit{true}$ and
2. $\forall (u, v) \in E \cdot \mathcal{M}(u) \wedge [\![\tau(u, v)]\!] \Rightarrow \mathcal{M}(v)'$.

If ψ is a well-labeling of $\mathcal{A}$, we say that $\mathcal{A}$ is *well-labeled*. □

We also define the following key properties of ARGs:

Definition 2.3 (Other Properties of ARGs)**.** Let $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$ be an ARG of a program $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$.

- $\mathcal{A}$ is *safe* if and only if for all $v \in V$ such that $\nu(v) = \mathsf{err}$, $\psi(v) \equiv \mathit{false}$.
- $\mathcal{A}$ is *complete* if and only if for all uncovered nodes u, for all $(\nu(u), T, \ell) \in \delta$, there exists an edge $(u, v) \in E$ such that $\nu(v) = \ell$ and $\tau(u, v) = T$. □

Using the above properties of ARGs, the following theorem shows how to prove safety of a program P by constructing an abstract reachability graph $\mathcal{A}$ of P. Informally, the labeling of a safe, complete, well-labeled ARG $\mathcal{A}$ of P implicitly encodes a safe inductive invariant of P, and thus implies its safety. We state the following theorem to demonstrate how ARGs can be used to prove program safety. This theorem is a generalization of [McM06, Theorem 1] to ARGs. A detailed proof is available in Appendix B.

Theorem 2.1 (Program Safety)**.** *If there exists a safe, complete, and well-labeled ARG for a program P, then P is safe.*

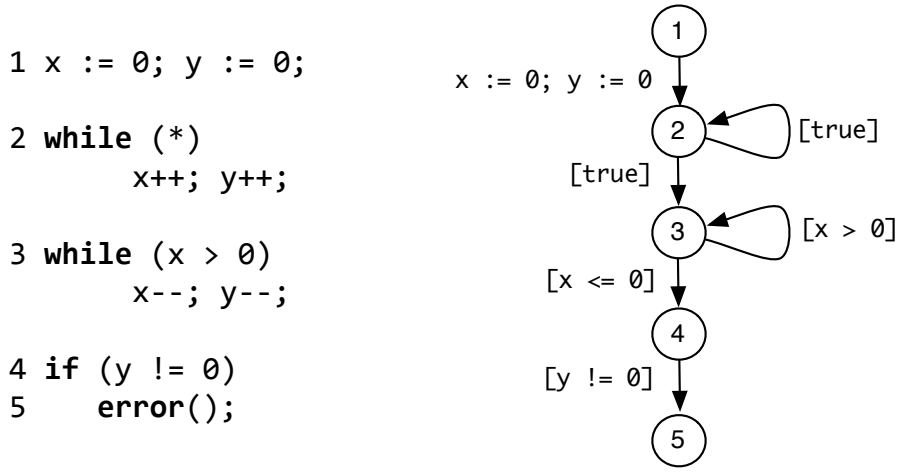
Proof. (*sketch*) This follows from [McM06, Theorem 1]. Suppose we are given a safe, complete, well-labeled ARG $\mathcal{A}$ of a program P. Then,

$$Inv = \{\ell \mapsto I_\ell \mid \ell \in \mathcal{L}\},$$

$$\text{where } I_\ell = \bigvee \{\psi(v) \mid v \in V \text{ and } \nu(v) = \ell \text{ and } v \text{ is uncovered}\},$$

is a safe inductive invariant of P. □

Example 2.1. *Figure 2.1 shows a program and its control-flow graph representation, where $*$ denotes non-deterministic choice. Our goal is to prove that location 5 (`error()`) is unreachable. By definition, an ARG may be a tree unrolling of the control-flow graph of the program or even a DAG unrolling. In Figure 2.2, we show two possible ARGs for that program. Each node v_i relates to program location i, i.e., $\nu(v_i) = i$, where program locations are the set $\{1, \ldots, 5\}$. The primes are used to distinguish between nodes relating to the same location. The dotted arrows are added for convenience to demonstrate covering,*



(a) (b)

Figure 2.1: Example program and its control-flow-graph representation.

e.g., the label of node v_2' is subsumed by the label of node v_2. Therefore, v_2' is covered by v_2. For clarity, we omit labels of some covered nodes. Note that both ARGs (a) and (b) are safe, since exit locations are labeled by false; well-labeled, since labels satisfy Definition 2.2; and complete, since every uncovered node has edges to all possible successor locations.

2.3 Iteration Strategy

A *Weak Topological Ordering* WTO [Bou93] of a directed graph $G = (V, E)$ is a well-parenthesized total-order, denoted $\prec$, of V without two consecutive "(" such that for every edge $(u, v) \in E$:

$$(u \prec v \wedge v \notin \omega(u)) \vee (v \preceq u \wedge v \in \omega(u)),$$

where elements between two matching parentheses are called a *(WTO-)component*, the first element of a component is called a *head*, and $\omega(v)$ is the set of heads of all components containing v.

We define two operations on a WTO, WtoNext and WtoExit:

- Let $v \in V$, and U be the innermost component that contains v in the WTO. We write WtoNext(v) for an element $u \in U$ that immediately follows v, if it exists, and for the head of U otherwise.
- Let U_v be a component with head v. First, suppose that U_v is a subcomponent of some component U. If there exists a $u \in U$ such that $u \notin U_v$ and u is the first element in the total-order such that $v \prec u$, then WtoExit(v) $= u$. Otherwise, WtoExit(v) $= w$, where w is the head of U. Second, suppose that U_v is not a subcomponent of any other component, then WtoExit(v) $= u$, where u is the first element in the total-order such that $u \notin U_v$ and $v \prec u$. Intuitively, if the WTO represented nodes/locations in a program's control-flow graph, then WtoExit(v) is the first control location visited after exiting the loop headed by v, where the locations in the body of the loop are in a WTO-component headed by v.

For example, for the program in Figure 2.1(b), a WTO of the control locations is

$$\ell_1\ (\ell_2)\ (\ell_3)\ \ell_4\ \ell_5,$$

where ℓ_2 is the head of the component comprising the first while loop and ℓ_3 is the head of the component comprising the second loop. $\textsc{WtoNext}(\ell_2) = \ell_2$ and $\textsc{WtoExit}(\ell_2) = \ell_3$. Note that WtoNext and WtoExit are partial functions and we only use them where they have been defined.

A detailed description of how to compute WTOs is available in [Bou93]. For our purposes here, we only require their definition.

2.4 Classical and Sequence Interpolants

We assume that all *formulas* are in some interpreted first-order theory $\mathcal{T}$, e.g., linear integer arithmetic. Given a formula A, we use $FV(A)$ to denote the set of free variables appearing in A. Given a sequence of formulas $A_1, \ldots, A_n$, we use $FV(A_1, \ldots, A_n)$ to denote $\bigcup_{i\in[1,n]} FV(A_i)$. We use the terms *validity* and *satisfiability* as is standard for first-order logic.

Interpolants Given two formulas A and B such that $A \wedge B$ is unsatisfiable, an *interpolant* for the pair (A, B) is a formula I such that

1. $A \Rightarrow I$ is valid,
2. $I \Rightarrow \neg B$ is valid, and
3. $FV(I) \subseteq FV(A) \cap FV(B)$

In other words, an interpolant is a formula that is (1) implied by A, (2) unsatisfiable with B, and (3) over the free variables that are shared by A and B. In some quantifier-free theories relevant for encoding program semantics, like linear real arithmetic and propositional logic, a quantifier-free interpolant always exists for a pair of unsatisfiable formulas (A, B).

Example 2.2. *Let $A \equiv x = y + z \wedge y \geqslant 0 \wedge z > 5$ and $B \equiv x \leqslant -10$. Treating all variables as integer variables, it is obvious that $A \wedge B$ is unsatisfiable. One possible interpolant I for (A, B) is $x \geqslant 0$. Note that I is only over the variable x that is shared between A and B.*

Sequence Interpolants *Sequence interpolants* [HJMM04, McM06] extend interpolants from a pair of formulas (A, B) to a sequence of formulas $A_1, A_2, \ldots, A_n$. Assuming that $\bigwedge_{i\in[1,n]} A_i$ is unsatisfiable, a sequence of interpolants for $A_1, \ldots, A_n$ is a sequence of formulas $I_1, \ldots, I_{n+1}$ such that

1. $I_1 \equiv \mathit{true}$,
2. $I_{n+1} \equiv \mathit{false}$,
3. $\forall i \in [1, n] \cdot I_i \wedge A_i \Rightarrow I_{i+1}$, and
4. $\forall i \in [2, n] \cdot FV(I_i) \subseteq FV(A_1, \ldots, A_{i-1}) \cap FV(A_i, \ldots, A_n)$.

From this definition, we see that for all $i \in [1, n+1]$, I_i is an interpolant for the pair of formulas

$$\left(\bigwedge_{j \in [1, i-1]} A_j, \bigwedge_{j \in [i, n+1]} A_j \right).$$

The following example illustrates sequence interpolants.

Example 2.3. *Let A_1, A_2, A_3 be a sequence of formulas, where*

$$A_1 \equiv x_1 \geqslant 1$$
$$A_2 \equiv x_2 = x_1$$
$$A_3 \equiv x_2 = 0$$

$A_1 \wedge A_2 \wedge A_3$ is unsatisfiable. One possible sequence of interpolants is

$$I_1 \equiv \mathit{true}$$
$$I_2 \equiv x_1 \geqslant 1$$
$$I_3 \equiv x_2 \neq 0$$
$$I_4 \equiv \mathit{false}$$

Consider the interpolant I_2; it is over the only variable, x_1, that is shared between the two formulas A_1 and $A_2 \wedge A_3$. Note also that $I_2 \wedge A_2 \Rightarrow I_3$ is valid.

2.5 Predicate Abstraction

Predicate abstraction [GS97] is an abstract domain promoted by Graf and Saïdi as a technique for building a finite state abstraction of an infinite state system or a system with a prohibitively large state space. Soon after its debut, predicate abstraction found its way into software model checking, as successfully implemented by the Slam [BR01] project. In Slam, predicate abstraction was used to build a Boolean (finite domain) program from an infinite state C program by only tracking a few program facts (predicates) and abstracting away the rest of the program. Constructing the abstraction is an expensive step that requires theorem proving. To that end, predicate abstraction techniques with varying precision have been proposed.

Let $\mathsf{Preds} = \{P_1, \ldots, P_n\}$ be a set of predicates, where a predicate P_i is a first-order formula over the variables of the program in question. Most verification tools use predicates in linear or bitvector arithmetic, e.g., a predicate $x > 0$ tracks whether the variable x is greater than 0.

Predicate abstraction over-approximates the results of strongest postconditions over program statements. Let T be a program statement of program P with variables Var, and φ be a formula over Var. Cartesian and Boolean predicate abstraction are defined as follows:

- *Cartesian abstraction*: The Cartesian abstraction, $\mathsf{CPost}(\varphi, T)$, of the strongest postcondition $\mathsf{SP}(\varphi, T)$ is the formula

$$\bigwedge\{\neg P_i \mid P_i \in \mathsf{Preds},\ \mathsf{SP}(\varphi, T) \Rightarrow \neg P_i\} \wedge \bigwedge\{P_i \mid P_i \in \mathsf{Preds},\ \mathsf{SP}(\varphi, T) \Rightarrow P_i\}.$$

In other words, for each predicate $P_i \in \mathsf{Preds}$, if it is implied by $\mathsf{SP}(\varphi, T)$ then it is conjoined to the result (and similarly for the negation of each predicate). It is easy to see that to compute CPost, Cartesian predicate abstraction requires at most $2n$ calls to a theorem prover, where n is the number of predicates.

- *Boolean abstraction*: Whereas Cartesian abstraction is *efficient*—requiring a linear number of calls to a theorem prover in the size of the predicate set—it may be imprecise, as it does not find the strongest over-approximation of $\mathsf{SP}(\varphi, T)$ that can be described using Preds. Therefore, using Cartesian abstraction might lead to many refinements in a CEGAR loop in order to check for spurious counterexamples.

  Boolean predicate abstraction, on the other hand, is an aggressive abstraction that computes the strongest possible Boolean combination of Preds that over-approximates $\mathsf{SP}(\varphi, T)$, denoted $\mathsf{BPost}(\varphi, T)$. Specifically, given the set of sets of predicates $2^{\mathsf{Preds} \cup \mathsf{Preds}^\neg}$, where $\mathsf{Preds}^\neg = \{\neg P_i \mid P_i \in \mathsf{Preds}\}$, Boolean abstraction finds the strongest formula

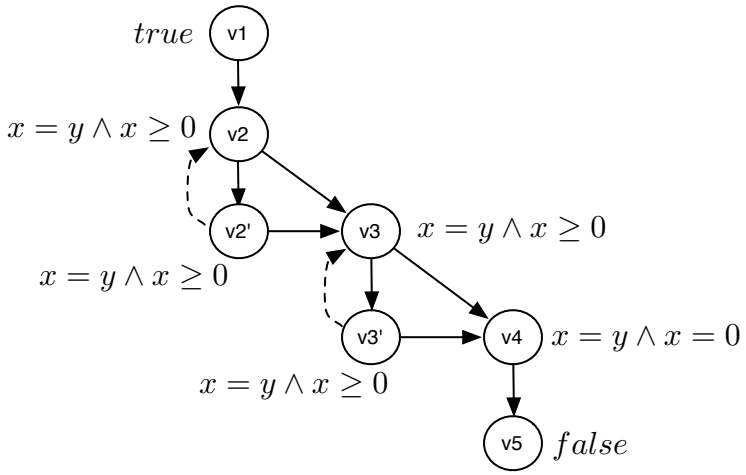
$$\bigvee_{P' \in S} \left(\bigwedge_{p \in P'} p \right),$$

  where $S \subseteq 2^{\mathsf{Preds} \cup \mathsf{Preds}^\neg}$, that is implied by $\mathsf{SP}(\varphi, T)$. In the worst case, one might require an exponential number of calls to the theorem prover, in the size of Preds, to compute $\mathsf{BPost}(\varphi, T)$.

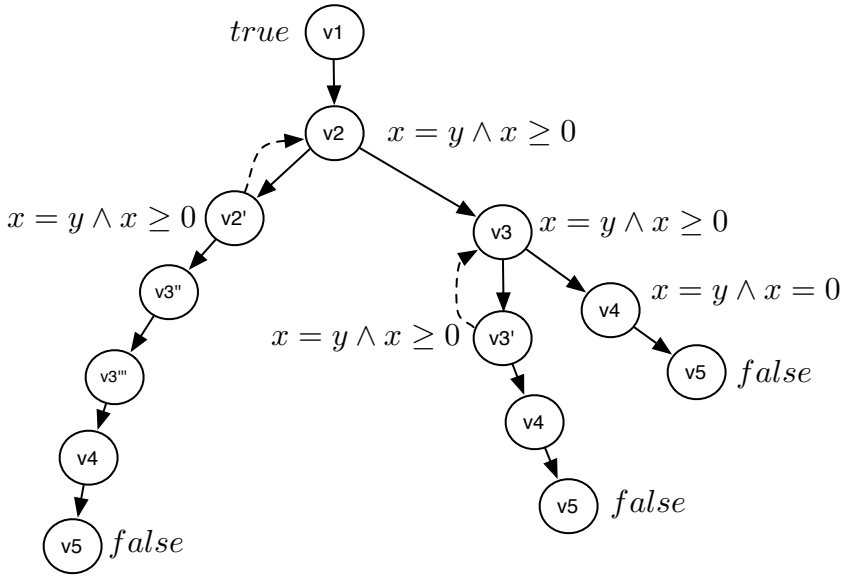
Example 2.4. *Let T be the sequence of two statements $\mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1$ in a program where the only variables are $\mathtt{x}$ and $\mathtt{y}$. Let $\varphi \equiv x = y \wedge x \geqslant 0$ and the predicate set $\mathsf{Preds} = \{x = y, x \geqslant 0\}$. To compute a Cartesian abstraction of $\mathsf{SP}(\varphi, T)$ over Preds, a verification tool typically makes the following four queries to a theorem prover, e.g., an SMT solver:*

1. *Is $\mathsf{SP}(\varphi, T) \Rightarrow x = y$ valid?*
2. *Is $\mathsf{SP}(\varphi, T) \Rightarrow x \geqslant 0$ valid?*
3. *Is $\mathsf{SP}(\varphi, T) \Rightarrow x \neq y$ valid?*
4. *Is $\mathsf{SP}(\varphi, T) \Rightarrow x < 0$ valid?*

Since only the first formula is valid, $\mathsf{CPost}(\varphi, T)$ returns the formula $x = y$.



(a)



(b)

Figure 2.2: Safe, complete, and well-labeled ARGs for the program in Figure 2.1.

Chapter 3

Verification with DAG Interpolants

3.1 Introduction

In this chapter, we introduce the concept of *Directed Acyclic Graph (DAG) interpolants*. DAG interpolants extend the concept of an interpolant between two formulas, or a sequence of formulas, to a set of formulas spatially arranged in a DAG structure. Given a technique for computing DAG interpolants, we can compute proofs for a set of program paths succinctly encoded as a DAG, where every path through the DAG represents a program path.

We then present (what we believe to be) an elegant transformation that encodes the DAG as a sequence of formulas, and reduces the problem of computing DAG interpolants to that of computing sequence interpolants. Effectively, we *linearize* the DAG: we view the program represented by the DAG, with all of its branching, as if it is a sequence of instructions (i.e., a single path). Thus, we use off-the-shelf sequence interpolation procedures within highly-efficient SMT solvers to compute DAG interpolants.

Armed with our procedure for computing DAG interpolants, we show how they can be used for verification of programs with loops. Specifically, we present a simple declarative procedure that, given a program, constructs a DAG-shaped abstract reachability graph which can be labeled using DAG interpolants.

Contributions

We summarize this chapter's contributions as follows:

- We extend the classical notion of an interpolant between two formulas to a set of formulas spatially organized as a directed acyclic graph.
- We present a technique for computing DAG interpolants by reducing the problem to sequence interpolation, thus enabling reuse of existing interpolation algorithms and implementations.
- We describe how DAG interpolants can be used to construct an interpolation-based verification technique by unrolling control-flow graphs of programs into DAGs.

Organization

This chapter is organized as follows:

- In Section 3.2, we formalize the concept of DAG interpolants.
- In Section 3.3, we present a procedure for computing DAG interpolants.
- In Section 3.4, we describe a simple and declarative verification procedure that uses DAG interpolants for computing safe inductive invariants.
- In Section 3.5, we place DAG interpolants within the growing mass of research on interpolant generation for verification.
- Finally, we conclude with a summary of the chapter in Section 3.6.

3.2 Graph Interpolation

In this section, we formally present DAG interpolants, a new form of interpolants that subsumes sequence interpolants (see Chapter 2).

DAG interpolants extend sequence interpolants to a set of formulas annotating edges of a directed acyclic graph. We also show that sequence interpolants are a special case of DAG interpolants.

In this chapter, we write

- F for the set of all possible formulas;
- $G = (V, E, v^{en}, v^{ex})$ for a DAG with an entry node $v^{en} \in V$ and an exit node $v^{ex} \in V$, where v^{en} has no predecessors, v^{ex} has no successors, and every node $v \in V$ lies on a (v^{en}, v^{ex})-path (a path from v^{en} to v^{ex});
- $desc(v)$ and $anc(v)$ for the sets of edges that can reach and are reachable from a node $v \in V$, respectively; and
- $\mathcal{L}_E : E \to F$ for a map from edges to formulas.

We call $\mathcal{L}_E$ an *edge labeling*. We use $\mathcal{L}_E(v_i, v_j)$ to denote the formula labeling the edge $(v_i, v_j) \in E$.

Definition 3.1 (DAG Interpolants (DItp))**.** Let G and $\mathcal{L}_E$ be as defined above. A DAG Interpolant for G and $\mathcal{L}_E$ is a map $\text{DItp} : V \to F$ such that

1. $\forall (v_i, v_j) \in E \cdot \text{DItp}(v_i) \wedge \mathcal{L}_E(v_i, v_j) \Rightarrow \text{DItp}(v_j)$,
2. $\text{DItp}(v^{en}) \equiv \mathit{true}$,
3. $\text{DItp}(v^{ex}) \equiv \mathit{false}$, and
4. $\forall v_i \in V \cdot FV(\text{DItp}(v_i)) \subseteq \left(\bigcup_{e \in desc(v_i)} FV(\mathcal{L}_E(e))\right) \cap \left(\bigcup_{e \in anc(v_i)} FV(\mathcal{L}_E(e))\right)$. □

Roughly speaking, if we take any path $v_1, \ldots, v_i, \ldots, v_n$ through the DAG, where $v_1 = v^{en}$ and $v_n = v^{ex}$, the DAG interpolant $\text{DItp}(v_i)$ at node v_i is an interpolant for the pair of formulas

$$\left(\bigwedge_{j \in [1, i-1]} \mathcal{L}_E(v_j, v_{j+1}), \bigwedge_{j \in [i, n-1]} \mathcal{L}_E(v_j, v_{j+1}) \right).$$

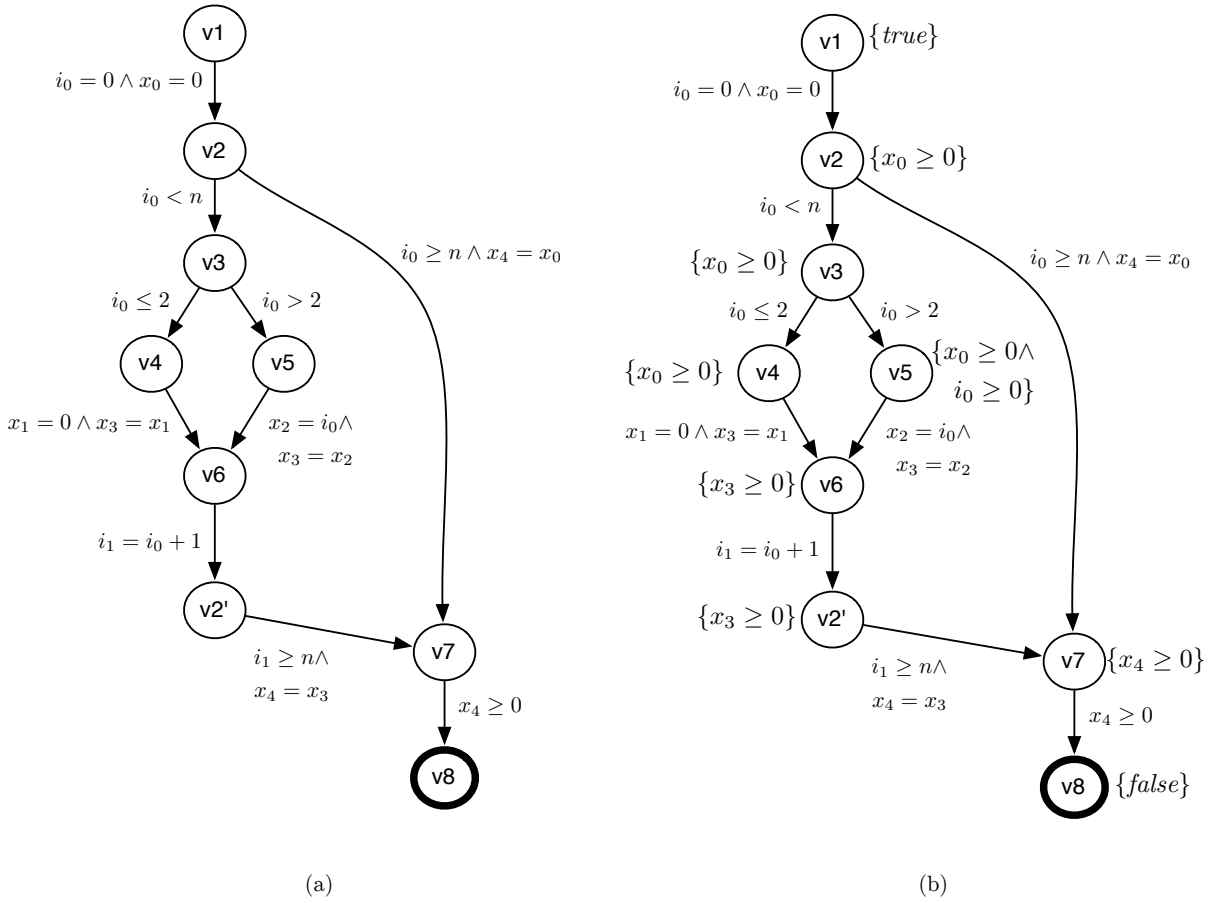


Figure 3.1: Example illustrating DAG interpolants.

Note that this is true assuming universal quantification of variables in $\text{DITP}(v_i)$ that are not shared between the A and B formulas.

From this observation, we note that if we restrict Definition 3.1 to DAGs where each node has at most one branch, i.e., the DAG is a single path from v^{en} to v^{ex}, the definition of DAG interpolants reduces to that of sequence interpolants (see Section 2.4). Thus, sequence interpolants are a special case of DAG interpolants. We illustrate DAG interpolants in the following example:

Example 3.1. *Consider the DAG in Figure 3.1(a), where node $v_1 = v^{en}$ and node $v_8 = v^{ex}$. The edges of the DAG are labeled by the formulas in $\mathcal{L}_E$. Figure 3.1(b) shows the same DAG with its nodes annotated by some DAG interpolants,* DITP*, in curly braces. Note that $\text{DITP}(v_1) = true$ and $\text{DITP}(v_8) = false$, as per Definition 3.1. Consider, for instance, the edge (v_4, v_6); we notice that*

$$\text{DITP}(v_4) \wedge \mathcal{L}_E(v_4, v_6) \Rightarrow \text{DITP}(v_6),$$

as per condition 1 in Definition 3.1. Also, note that for each $v \in V$, the variables appearing in $\text{DITP}(v)$ are only those that appear both on the edges of the subgraph that can reach v and the subgraph that is reachable from v, as per condition 4 in Definition 3.1.

3.3 Computing DAG Interpolants

In this section, we present a procedure for computing DAG interpolants. As is the case in most interpolation procedures, the first step is to prove unsatisfiability of the given formulas, after which we "mine" the refutation proof for interpolants. In classical interpolants, we prove unsatisfiability of a pair (A, B). In sequence interpolants, we prove unsatisfiability of the conjunction of all formulas in the sequence. In the case of DAG interpolants, we need to show that every path through the DAG is unsatisfiable. We do so using a *DAG Condition* formula.

DAG Condition Given a DAG G and edge labeling $\mathcal{L}_E$, we define DAGCond as follows:

$$\text{DAGCond}(G, \mathcal{L}_E) \equiv c_{v_1} \wedge \mu_1 \wedge \cdots \wedge \mu_n, \tag{3.1}$$

$$\text{where } \mu_i \equiv c_{v_i} \Rightarrow \bigvee_{(v_i,w)\in E} (c_w \wedge \mathcal{L}_E(v_i, w))\,; \tag{3.2}$$

$v_1 = v^{en}$; $v_1, \ldots, v_n$ is a sequence of all nodes in $V \setminus \{v^{ex}\}$ ordered by $\sqsubseteq_t$, a fixed linearization of the topological ordering of the nodes in V; and c_{v_i} is a fresh Boolean *control variable* representing the node v_i.

This is a standard encoding of all paths in a DAG that start in v^{en} and end in v^{ex}. Therefore, $\text{DAGCond}(G, \mathcal{L}_E)$ is unsatisfiable if and only if for every (v^{en}, v^{ex})-path through the DAG, the conjunction of all edge labels along the path is unsatisfiable.

Example 3.2. *Consider the DAG in Figure 3.1(a); call it G and its edge labeling $\mathcal{L}_E$. Assume we pick the following total (and topological) order for the nodes V of G: $v_1, v_2, v_3, v_4, v_5, v_6, v_2', v_7, v_8$. Then,* $\text{DAGCond}(G, \mathcal{L}_E) \equiv c_{v_1} \wedge \mu_1 \wedge \cdots \wedge \mu_7$, *where*

$$\begin{aligned}
&\mu_1 \equiv c_{v_1} \Rightarrow (i_0 = 0 \wedge x_0 = 0 \wedge c_{v_2})\\
&\mu_2 \equiv c_{v_2} \Rightarrow ((i_0 < n \wedge c_{v_3}) \vee (i_0 \geqslant n \wedge x_4 = x_0 \wedge c_{v_7}))\\
&\mu_3 \equiv c_{v_3} \Rightarrow ((i_0 \leqslant 2 \wedge c_{v_4}) \vee (i_0 > 2 \wedge c_{v_5}))\\
&\mu_4 \equiv c_{v_4} \Rightarrow (x_1 = 0 \wedge x_3 = x_1 \wedge c_{v_6})\\
&\mu_5 \equiv c_{v_5} \Rightarrow (x_2 = i_0 \wedge x_3 = x_2 \wedge c_{v_6})\\
&\mu_6 \equiv c_{v_6} \Rightarrow (i_1 = i_0 + 1 \wedge c_{v_2'})\\
&\mu_2' \equiv c_{v_2'} \Rightarrow (i_1 \geqslant n \wedge x_4 = x_3 \wedge c_{v_7})\\
&\mu_7 \equiv c_{v_7} \Rightarrow (x_4 \geqslant 0 \wedge c_{v_8})
\end{aligned}$$

DAG Interpolants from Sequence interpolants We now describe how to compute DAG interpolants via a transformation of sequence interpolants over the formulas constituting $\text{DAGCond}(G, \mathcal{L}_E)$.

Let $I_1, \ldots, I_{n+1}$ be some sequence interpolants for the sequence of formulas $(c_{v_1} \wedge \mu_1), \mu_2, \ldots, \mu_n$ constituting $\text{DAGCond}(G, \mathcal{L}_E)$, as defined in formula 3.1 above. From the sequence interpolants $I_1, \ldots, I_{n+1}$, we want to *extract* DAG interpolants for G and $\mathcal{L}_E$. Specifically, for each node v_i, our goal is to extract $\text{DItp}(v_i)$ from the interpolant I_i. The following example demonstrates the difficulty this process entails.

Example 3.3. *Continuing Example 3.2, consider node v_6 from Figure 3.1. We want to extract (for instance) $x_3 \geqslant 0$ from I_6. I_6 is the sequence interpolant that comes right before μ_6, i.e., the one that*

satisfies the condition $I_6 \wedge \mu_6 \Rightarrow I'_2$*, by definition of sequence interpolants. The interpolant* I_6 *can be, for instance,*

$$(c_{v_6} \wedge x_3 \geqslant 0) \vee (c_{v_7} \wedge x_4 \geqslant 0).$$

Note that the interpolant I_6 *can contain both control variables (e.g.,* c_{v_6}*) and variables that should not appear in the DAG interpolant at node* v_6*, namely, the variable* x_4*. Therefore, we need a way of extracting the "useful" part of* I_6*: the one we need for* $\text{DITP}(v_6)$*.*

The following procedure takes an interpolant I_i and returns a formula I'_i. When applied to the sequence of interpolants $I_1, \ldots, I_{n+1}$, it results in DAG interpolants for G and $\mathcal{L}_E$.

$$\text{CLEAN}(I_i) \equiv \ \forall\{x \mid x \in FV(I_i) \wedge \neg \mathit{inScope}(x, v_i)\} \cdot \forall\{c_{v_j} \mid v_j \in V\} \cdot I[c_{v_i} \leftarrow \mathit{true}],$$

where $\mathit{inScope}(x, v_i)$ is is true if and only if $x \in \left(\bigcup_{e \in desc(v_i)} FV(\mathcal{L}_E(e))\right) \cap \left(\bigcup_{e \in anc(v_i)} FV(\mathcal{L}_E(e))\right)$. We say that x is *out-of-scope* at node v_i if $\neg \mathit{inScope}(x, v_i)$ holds.

Example 3.4. *Continuing Example 3.3,*

$$I'_6 = \text{CLEAN}(I_6) = \forall x_4, c_{v_7} \cdot I_6[c_{v_6} \leftarrow \top] = x_3 \geqslant 0.$$

Note that x_4 *is universally quantified since it is not within the set of variables that are allowed to appear at node* v_6*;* c_{v_6} *is set to true because we want to "focus" on the label of node* v_6*; and* c_{v_7} *is universally quantified because we do not care about the label of node* v_7*—just* v_6*.*

By definition, $\text{CLEAN}(I_i)$ is a formula over the variables that are allowed to appear at node v_i. Specifically, CLEAN eliminates, via universal quantification and substitution, all control variables and out-of-scope variables at node v_i. Theorem 3.1 states that the transformed interpolants resulting from applying CLEAN to sequence interpolants are indeed DAG interpolants.

Theorem 3.1. *Let* $I'_k = \text{CLEAN}(I_k)$ *for all* $k \in [1, n+1]$*.* $\text{DITP} = \{v_i \mapsto I'_i | i \in [1, n+1]\}$ *are DAG interpolants for* G *and* $\mathcal{L}_E$*, as per Definition 3.1.*

Proof. We proceed by showing that DITP satisfies the four conditions in Definition 3.1.

- *Condition 1*: We want to show that for any two vertices v_i and v_j such that $(v_i, v_j) \in E$, we have $\text{DITP}(v_i) \wedge \mathcal{L}_E(v_i, v_j) \Rightarrow \text{DITP}(v_j)$. By definition, for any two such vertices, we know that $v_i \sqsubset_t v_j$, and therefore interpolant I_i is before I_j in the computed sequence interpolants. It follows

by definition of DAGCOND and sequence interpolants that:

$$
\begin{aligned}
& I_i \wedge \mu_i \wedge \cdots \wedge \mu_{j-1} \Rightarrow I_j \\
& \quad \text{(set } c_{v_i} \text{ to } \mathit{true} \text{ and simplification)} \\
\Rightarrow \quad & I_i[c_{v_i} \leftarrow \mathit{true}] \wedge c_{v_j} \wedge \mathcal{L}_E(v_i, v_j) \wedge \mu_{i+1} \wedge \cdots \wedge \mu_{j-1} \Rightarrow I_j \\
& \quad \text{(let } \Pi = \{c_{v_{i+1}}, \ldots, c_{v_{j-1}}\}) \\
\Rightarrow \quad & I_i[c_{v_i} \leftarrow \mathit{true}, \Pi \leftarrow \mathit{false}] \wedge c_{v_j} \wedge \mathcal{L}_E(v_i, v_j) \Rightarrow I_j \\
& \quad \text{(set } c_{v_j} \text{ to } \mathit{true}) \\
\Rightarrow \quad & I_i[\Pi \leftarrow \mathit{false}, c_{v_i} \leftarrow \mathit{true}, c_{v_j} \leftarrow \mathit{true}] \wedge \mathcal{L}_E(v_i, v_j) \Rightarrow I_j[c_{v_j} \leftarrow \mathit{true}] \\
& \quad \text{(use } (\forall x.f) \Rightarrow f) \\
\Rightarrow \quad & I'_i \wedge \mathcal{L}_E(v_i, v_j) \Rightarrow I_j[c_{v_j} \leftarrow \mathit{true}] \\
& \quad \text{(out-of-scope variables of } v_j \text{ are not in the antecedent)} \\
\Rightarrow \quad & I'_i \wedge \mathcal{L}_E(v_i, v_j) \Rightarrow I'_j
\end{aligned}
$$

Therefore, we have $\text{DITP}(v_i) \wedge \mathcal{L}_E(v_i, v_j) \Rightarrow \text{DITP}(v_j)$, since $\text{DITP}(v_i) = I'_i$ and $\text{DITP}(v_j) = I'_j$.

- *Condition 2*: Follows trivially from the fact that $I_1 \equiv \mathit{true}$, by definition of sequence interpolants.
- *Condition 3*: Follows trivially from the fact that $I_{n+1} \equiv \mathit{false}$, by definition of sequence interpolants.
- *Condition 4*: By definition of $\text{CLEAN}(I_i)$, in the resulting formula I'_i, all control variables are bound by the universal quantifier or are replaced by constants. Similarly, all non-control variables that are not in

$$\left(\bigcup_{e \in desc(v_i)} FV(\mathcal{L}_E(e))\right) \cap \left(\bigcup_{e \in anc(v_i)} FV(\mathcal{L}_E(e))\right)$$

are bound by the universal quantifier.

□

In summary, we have shown how to compute DAG interpolants using a three-step process:

1. encode DAG as a sequence of formulas, a DAG condition, where each formula in the sequence encodes one of the nodes and the edges emanating from it;
2. compute a sequence of interpolants for the DAG condition; and,
3. finally, transform sequence interpolants into DAG interpolants.

3.4 Verification with DAG Interpolants

In this section, we demonstrate how DAG interpolants can be utilized for proving program safety. To that end, we present a simple declarative procedure that uses DAG interpolants to label an abstract

reachability graph (ARG) of a given program (see Definition 2.1). We illustrate the process through an example as we present it. In Chapter 4, we present an operational (and more detailed) verification procedure. Our goal in this chapter is to demonstrate, generically, how DAG interpolants can be used for verifying safety properties of programs.

In Chapter 2, we formally defined and illustrated ARGs as a mechanism for proving program safety. Specifically, we showed that a safe, complete, well-labeled ARG $\mathcal{A}$ of a program P implies that the program is safe (by Theorem 2.1). To prove program safety, we proceed in two steps:

1. Construct an ARG $\mathcal{A}$ of a given program P.

2. Label nodes of $\mathcal{A}$ (i.e., define ψ) such that the result is a safe, complete, well-labeled ARG. We demonstrate how this can be achieved with DAG interpolants.

Abstract Reachability Graphs of Programs Given a program $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$, we first construct a DAG-shaped ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$ of P. We assume that we have a procedure that constructs well-labeled, complete, but not necessarily safe ARGs. We also assume assume that one and only one node in the ARG maps to the error location in the program. That is, we assume that there exists one and only one node $v \in V$ such that $\nu(v) = \mathsf{err}$—we use v_{err} to denote such node. The following example shows a program and one of its possible ARGs.

Example 3.5. *Consider the program $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ in Figure 3.2(a). The locations $\mathcal{L}$ of P are the set of integers $\{1, \ldots, 9\}$. The error location err is 8. The instruction $\mathtt{x} := \mathtt{0}$ at location 4 is represented by the action*

$$(4, \mathtt{x} := \mathtt{0}, 6) \in \delta.$$

One possible ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$ for P is shown in Figure 3.2(b). The subscript of each node of $\mathcal{A}$ denotes the program location it maps to. For instance, $\nu(v_2) = 2$. The formula in curly braces beside each node v is its label $\psi(v)$. For instance, $\psi(v_1) = \mathit{true}$. Note that using true as the label of all nodes always results in a well-labeled ARG. For any edge $(v_i, v_j) \in E$ in this ARG, the edge label $\tau(v_i, v_j)$ is a program instruction T such that $(i, T, j) \in \delta$.

This ARG is well-labeled, because all nodes are labeled true and thus satisfy Definition 2.2; complete, because node v_2' is covered by node v_2 (as shown by the backwards dotted arrow); and unsafe, since node v_8, which maps to the error location, is not labeled by false.

Intuitively, $\mathcal{A}$ represents an unrolling of the control-flow graph of P, where the body of the while loop is allowed to execute at most once. This is similar to a BMC unrolling of a program [CKL04].

DAG Interpolants for Labeling ARGs Now that we have an ARG $\mathcal{A}$ of program P, we would like to find a labeling φ of its nodes such that it becomes well-labeled, safe, and complete. To do so, we use DAG interpolants.

We view an ARG $\mathcal{A}$ as a DAG $G = (V', E', v^{en}, v^{ex})$, where $v^{en} = v_{\mathsf{en}}$ and $v^{ex} = v_{\mathsf{err}}$. The sets of vertices and edges, V' and E', of G are the same as those in the ARG minus edges/vertices that cannot reach v_{err}. For example, for the ARG in Figure 3.2(b), edges (v_2', v_3') and (v_7, v_9) are not in E', and nodes v_3' and v_9 are not in V'.

We now need to compute an edge labeling, $\mathcal{L}_E$, for G that encodes the semantics of program instructions represented by the edges. For the purpose of presentation, we provide a simplified definition of our

encoding. In practice, we use the *Static single assignment* (SSA) form encoding defined in [GCS11]. Let

$$\mathsf{SVar} = \{x_v \mid x \in \mathsf{Var} \wedge v \in V'\}$$

be the set of variables that can appear in $\mathcal{L}_E$. That is, for each variable $x \in \mathsf{Var}$ and node $v \in V'$, we create a *symbolic variable* $x_v \in \mathsf{SVar}$. The map $\mathsf{SMap} : \mathsf{SVar} \to \mathsf{Var}$ associates each x_v with its program variable x. The following definition formalizes the process of encoding edge labels from instructions.

Definition 3.2 (Encoding edge labels $\mathcal{L}_E$)**.** For an edge $(u, v) \in E'$:

- If $\tau(u, v)$ is an assignment statement `x := E`, then

$$\mathcal{L}_E(u, v) = (x_v = E[x \leftarrow x_u]) \wedge \bigwedge \{y_v = y_u \mid y \in \mathsf{Var} \wedge y \neq x\}.$$

- If $\tau(u, v)$ is an assume statement `assume`(Q), then

$$\mathcal{L}_E(u, v) = Q[x \leftarrow x_u \mid x \in \mathit{var}(Q)] \wedge \bigwedge \{y_v = y_u \mid y \in \mathsf{Var}\},$$

  where $\mathit{var}(Q)$ is the set of variables appearing in Q. □

In other words, each assignment instruction to variable `x` is modelled as a formula that updates the value of x at the destination node, while maintaining the values of all other variables as they were at the source node (i.e., a frame condition). `assume` instructions constrain the values variables can take. For example, for an edge $(u, v) \in E$ such that $\tau(u, v)$ is `x := x + 1`, the edge label $\mathcal{L}_E(u, v)$ is

$$x_v = x_u + 1 \wedge y_v = y_u,$$

assuming $\mathsf{Var} = \{x, y\}$.

There are two points to note here

1. Our encoding results in a total onto map from satisfying assignments of DAGCond$(G, \mathcal{L}_E)$ to feasible program executions represented by paths from v_{en} to v_{err} (the node that maps to the error location) through the ARG.

2. As a result, if DAGCond$(G, \mathcal{L}_E)$ is unsatisfiable, we can compute DAG interpolants for G, from which we can extract a safe well-labeling of the ARG.

Note that DAG interpolants for G will be over the set of symbolic variables SVar. Thus, to extract a safe well-labeling for the ARG from DItp, we need to rename variables back to their original names. Specifically, we use the following simple transformation:

$$\psi = \{v \mapsto \text{DItp}(v)[x \leftarrow \mathsf{SMap}(x) \mid x \in \mathsf{SVar}] \mid v \in V'\} \cup \{v \mapsto \mathit{true} \mid v \in V \setminus V'\}$$

which replaces every symbolic variable x_v with its original variable x (using the map SMap). Additionally, nodes that are in the ARG but not in G are labeled by *true* (thus maintaining well-labeledness of the ARG).

Example 3.6. *Recall Example 3.1 illustrated in Figure 3.1. Figure 3.1(b) happens to show the DAG G resulting from the ARG in Figure 3.2(b). The edge labeling of G is a simplified version of our*

above encoding, to avoid too many extraneous constraints. Each variable x has a number of symbolic counterparts, x_i, where i is an integer subscript. The labels of the nodes in Figure 3.1(b) are DAG interpolants. By removing the subscripts from symbolic variables, we arrive at a safe, well-labeled ARG, shown in Figure 3.2(c). Our new labels also result in a complete ARG, and therefore we conclude that the program is correct: there is no execution that can reach the error location (location 8). From the labels of the ARG, we notice that the inductive invariant of the while loop (label of node v_2) is $x \geqslant 0$.

Summary In this section, we have shown how to compute a safe well-labeling of an ARG using DAG interpolants, but we have left a number of questions unanswered:

- Given a program P, how do we construct an ARG $\mathcal{A}$?
- What if DAG interpolants do not result in a complete ARG?

In Chapter 4, we answer these questions by showing how to systematically grow an abstract reachability graph and use DAG interpolants to label it. In addition, we demonstrate how to incorporate AB techniques within this IB framework to improve performance.

3.5 Related Work and Survey of Interpolation Techniques

In this section, we place DAG interpolants within the landscape of related work and provide an overview of interpolant generation techniques.

Interpolants from Resolution Proofs In his initial work on SAT-based model checking with interpolation [McM03], McMillan introduced an interpolation procedure for propositional logic. McMillan's procedure assumes existence of a resolution proof of unsatisfiability of a pair of formulas (A, B). By traversing the resolution proof and maintaining *partial interpolants*, an interpolant for (A, B) can be computed in time linear in the size of the proof. Within the verification and decision procedures communities, this resulted in a large number of papers extending McMillan's algorithm to more expressive theories and studying its properties.

In [McM04], McMillan introduced an interpolation procedure from refutation proofs for the theory of linear arithmetic and uninterpreted functions; this procedure was used in the Blast software model checker [HJMM04] for predicate discovery and the Impact software model checker for interpolation-based verification [McM06]. Jhala and McMillan [JM07] extended [McM04] for computing quantified interpolants of restricted form in the theory of arrays. A number of other works explored interpolation in the theory of bitvectors [KW07, Gri11], with the goal of enabling bit-precise encodings of program semantics.

For any pair of formulas (A, B) such that $A \wedge B$ is unsatisfiable, there can be a range of possible interpolants. An interpolating procedure computes one specific interpolant within a possibly infinite set of interpolants. The work of D'Silva et al. [DKPW10] studied the range of interpolants that can be computed from a given propositional resolution proof, characterizing them in terms of strength. Weissenbacher [Wei12] extended [DKPW10] to strength of interpolants in first-order proof systems and hyper-resolution proofs.

All of the above works on computing interpolants are orthogonal to the problem addressed in this chapter: computing DAG interpolants. Our proposed procedure reduces the problem to computing

sequence interpolants, a well-studied problem in the above-mentioned works. Thus, we can directly leverage advances in interpolation procedures for computing DAG interpolants.

A number of new forms of interpolants have also been recently proposed. Tree interpolants [MR13] define interpolants over a tree labeled with formulas. Tree interpolants are incomparable to DAG interpolants, though both subsume sequence interpolants. Disjunctive interpolants [RHK13b] generalize tree interpolants to interpolation between a formula and one of its subexpressions. Like tree interpolants, disjunctive interpolants are also incomparable with DAG interpolants [RHK13a].

Interpolants and Horn Clauses Recently, there has been growing interest in casting verification problems as solving Horn-like clauses. Interpolation can be utilized as a means for solving different classes of Horn clauses. Rümmer et al. [RHK13a] connect different forms of interpolation (classical, sequence, DAG, tree, etc.) to different classes of Horn clauses. For instance, they show that DAG interpolants subsume sequence interpolants and can be used for solving *linear non-recursive* Horn clauses. Gupta et al. [GPR11] present a specialized procedure for solving linear non-recursive Horn clauses for the combined theories of linear integer arithmetic and uninterpreted functions.

Interpolation-based Verification Techniques Interpolation-based verification has received a great deal of interest over the past few years. We delay our comparison with IB techniques and others to Chapter 4.

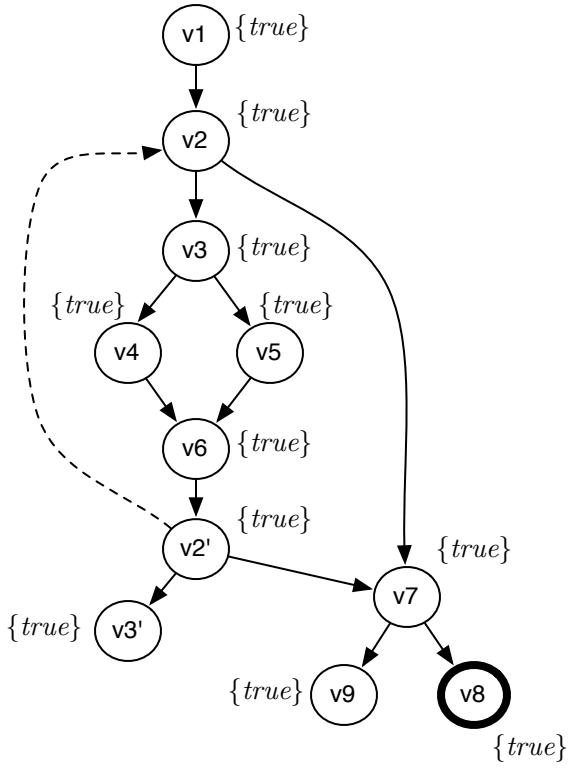
3.6 Conclusion

Encoding finite (bounded) program executions as formulas dates back to, at least, Cook's proof that 3SAT is NP-Complete [Coo71]. Later, King [Kin76] introduced symbolic execution with the goal of test generation and program exploration. Advances in SAT/SMT solving and bounded model checking revived interest in the area. Craig interpolants added a new dimension to symbolic encodings of bounded executions: they enabled inferring proofs of correctness for the unbounded case. In this chapter, we introduced a new form of interpolants, DAG interpolants, that allow us to examine multiple bounded paths through the program simultaneously through a DAG encoding. We showed that we can utilize the power and efficiency of modern SMT solvers (with their interpolation features) to compute a Hoare-style proof of a loop-free unrolling of a program, from which we can infer a proof of the whole program.

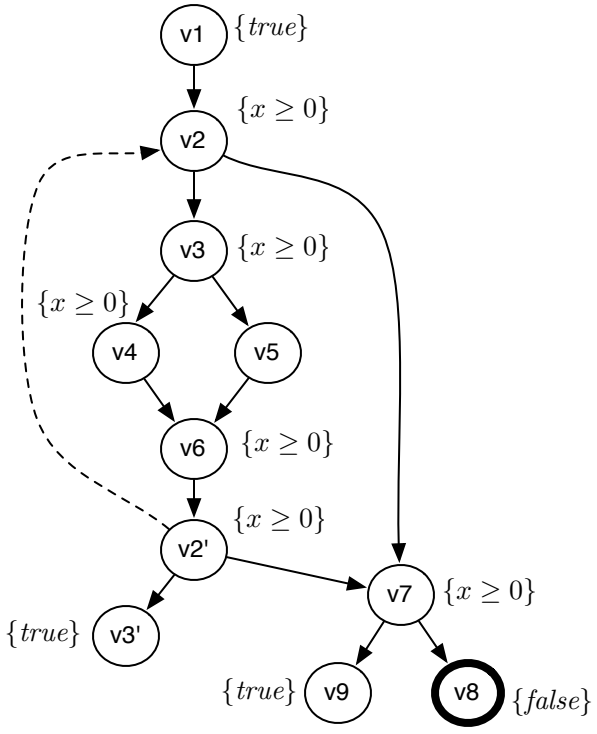
DAG interpolants generalize McMillan's sequence interpolants to sets of sequences encoded as a directed acyclic graph. As a result, we demonstrated how DAG interpolants can be used for software verification, in a style similar to McMillan's lazy abstraction with interpolants (LAWI). In comparison with LAWI, our procedure does not unroll the control-flow graph of the program into a tree; instead, it unrolls the program into a DAG, and uses DAG interpolants to hypothesize a safe inductive invariant. DAG interpolants allow us to avoid path explosion that could result from an explicit tree unrolling of the program by delegating the explosion to the SMT solver. In the rest of this dissertation, we describe efficient verification algorithms that utilize DAG interpolants, demonstrate their effectiveness, and extend them in various directions.

```
1 i := 0, x := 0;
2 while (i < n)
3    if (i <= 2)
4       x = 0;
     else
5       x := i;
6   i := i + 1;
7 if (x < 0)
8    error();
9 return;
```

(a)



(b)



(c)

Figure 3.2: Safe, complete, well-labeled ARG using DAG interpolants.

Chapter 4

Predicate Abstraction and Interpolation-based Verification

4.1 Introduction

In Chapter 1, we categorized automated verification techniques into abstraction-based (AB) and interpolation-based (IB), and discussed their advantages and disadvantages. In AB techniques, an abstract fixpoint computation is used to compute an inductive invariant for the program by *executing* an abstract version of the program, as defined by the abstract domain. On the other hand, IB techniques do not restrict the search for an inductive invariant by an abstract domain and do not perform a forward/backward fixpoint computation; instead, they operate by hypothesizing invariants from proofs of correctness of finite paths through a program's control-flow graph.

In this chapter, we present Ufo, an automated verification algorithm that combines AB and IB verification. Ufo is parameterized by the degree with which IB or AB drives the analysis.[1] From a technical perspective, Ufo makes a number of contributions:

- On one extreme, when Ufo is instantiated *without* any predicate abstract domain, it is an efficient implementation of the IB technique we presented in Chapter 3, where DAG interpolants are used to hypothesize safe inductive invariants.
- On the other extreme, Ufo can be instantiated in such a way that AB techniques drive the analysis, and DAG interpolants are simply used to add new predicates (in the CEGAR refinement phase) in case unsafe inductive invariants are computed.
- In the middle, Ufo can be instantiated as a hybrid IB/AB algorithm, where IB and AB techniques alternate and build on the results of each other.

All of these instantiations result in novel algorithms and allow us to evaluate different ends of the IB/AB spectrum. Ufo is implemented in the UFO_{app} verification tool and framework (see Chapter 7), in the LLVM compiler infrastructure [LA04]. Due to an unfortunate historical mistake, the Ufo algorithm and the UFO_{app} tool have the same name; to clearly distinguish between them, we always use a different font and the subscript *app* when we are referring to the tool. Our experimental evaluation of different

[1] The U in Ufo stands for under-approximation, O for over-approximation, and F for a function combining both.

Ufo instantiations on a suite of C programs demonstrates (1) the utility of our IB instantiation of Ufo and (2) the power of hybrid IB/AB instantiations in comparison with either extreme.

Contributions

We summarize this chapter's contributions as follows:

- We present a parameterized algorithm that integrates abstraction-based and interpolation-based verification techniques.
- We show how our algorithm can be instantiated into abstraction-based algorithms, the interpolation-based algorithm presented in Chapter 3, as well as novel hybrid algorithms that combine advantages of abstraction- and interpolation-based techniques.
- We evaluate the efficiency of concrete instantiations of our algorithm and show that hybrid IB/AB instantiations of the algorithm can outperform pure IB and AB techniques. Further, we show our DAG-interpolation-based technique (Chapter 3) outperforms an implementation of McMillan's original IB algorithm [McM03].

Organization

This chapter is organized as follows:

- In Section 4.2, we present the verification algorithm Ufo.
- In Section 4.3, we present an experimental evaluation of different instantiations of Ufo.
- In Section 4.4, we place Ufo within IB and AB techniques from the literature.
- Finally, in Section 4.5, we summarize the chapter.

4.2 The Ufo Algorithm

In this section, we present our parameterized verification algorithm, Ufo, and describe a range of possible instantiations. At a high level, Ufo alternates between two phases, one using interpolants to hypothesize a safe inductive invariant and one using an abstract fixpoint computation to compute an inductive invariant. Both phases share information by operating over the same data structure, an abstract reachability graph (Definition 2.1). The process continues until a safe inductive invariant or a counterexample is found.

4.2.1 Parameterized Algorithm

The Ufo algorithm takes a program $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ and determines whether it is safe or unsafe. The output of the algorithm is either an execution of P that ends in err, i.e., a counterexample, or a complete, well-labeled, safe ARG $\mathcal{A}$ of P, indicating that the program is safe.

The novelty of Ufo lies in its combination of IB and AB techniques. Figure 4.1 illustrates the two main states of Ufo:

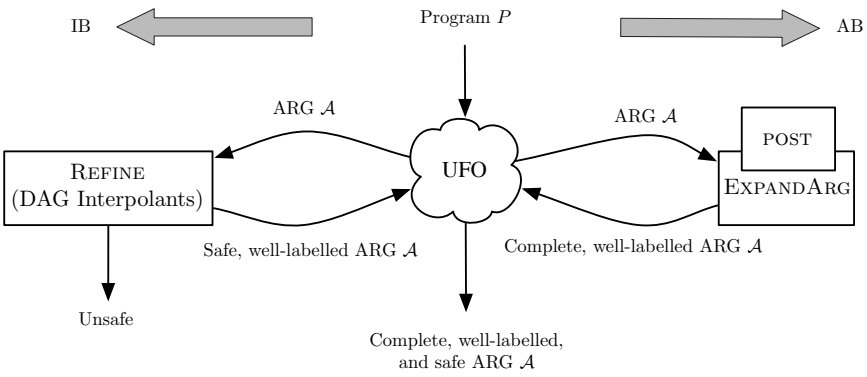


Figure 4.1: High level description of Ufo.

Algorithm 1 The Ufo Algorithm.

```
1:  function UfoMain(Program P)
2:      create node v_en
3:      ψ(v_en) ← true
4:      ν(v_en) ← en
5:      marked(v_en) ← true
6:      labels ← ∅
7:      while true do
8:          ExpandArg()
9:          if ψ(v_err) is UNSAT then
10:             return SAFE
11:         labels ← Refine()
12:         if labels = ∅ then
13:             return UNSAFE
14:         clear AH and FN

15: function GetFutureNode(ℓ ∈ L)
16:     if FN(ℓ) exists then
17:         return FN(ℓ)
18:     create node v
19:     ψ(v) ← true
20:     ν(v) ← ℓ
21:     FN(l) ← v
22:     return v

23: function ExpandNode(v ∈ V)
24:     if v has children then
25:         for all (v, w) ∈ E do
26:             FN(ν(w)) ← w
27:     else
28:         for all (ν(v), T, ℓ) ∈ δ do
29:             w ← GetFutureNode(ℓ)
30:             E ← E ∪ {(v, w)}
31:             τ(v, w) ← T
```

- *Exploring (AB)*: The exploration phase is an abstract fixpoint computation to compute an inductive invariant of P. Specifically, exploring constructs an ARG of P by unwinding the control-flow graph of P while *computing* node labels using an abstract post operator, Post. The result is always a complete, well-labeled ARG. Of course, the ARG might be unsafe due to imprecision in Post.

- *Generalizing (IB)*: Generalizing is done by computing (typically using DAG interpolants) a safe, well-labeling of the current ARG from a proof of infeasibility of execution paths to error nodes in the ARG. Of course, interpolants are not guaranteed to give a complete ARG.

By alternating between these two phases, Ufo combines AB and IB techniques.

The pseudo-code of Ufo is given in Algorithms 1 and 2. Function ExpandArg (Algorithm 2) is responsible for the exploration and Refine (line 11) for generalization. Note that Ufo is parameterized by Post (line 35). More precise Post makes Ufo more AB-like; less precise Post makes Ufo more IB-like.

Algorithm 2 Ufo's ExpandArg algorithm.

```
32: function ExpandArg
33:     v ← v_en
34:     while true do
35:         ExpandNode(v)
36:         if marked(v) then
37:             marked(v) ← false
38:             ψ(v) ← ⋁_{(u,v)∈E} Post(u, v)
39:             for all (v, w) ∈ E do
40:                 marked(w) ← true
41:         else if labels(v) bound then
42:             ψ(v) ← labels(v)
43:             for all {(v, w) ∈ E | labels(w) unbound} do
44:                 marked(w) ← true
45:         if v = v_err then break
46:         if ν(v) is head of a component then
47:             if ψ(v) ⇒ ⋁_{u∈AH(ν(v))} ψ(u) then
48:                 erase AH(ν(v)) and FN(ν(v))
49:                 l ← WtoExit(ν(v))
50:                 v ← FN(l)
51:                 erase FN(l)
52:                 for all {(v, w) ∈ E | ∄u ≠ v · (u, w) ∈ E} do
53:                     erase FN(ν(w))
54:                 continue
55:             add v to AH(ν(v))
56:         l ← WtoNext(ν(v))
57:         v ← FN(l)
58:         erase FN(l)
```

Main Loop UfoMain is the main function of Ufo.[2] It receives a program $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ as input and attempts to prove that P is safe (or unsafe) by constructing a complete, well-labeled, safe ARG for P (or by finding an execution to err). The function ExpandArg is used to construct an ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$ for P. By definition, it always constructs a complete, well-labeled ARG. Line 8 of UfoMain checks if the result of ExpandArg is a safe ARG by checking whether the label on the node v_{err} is satisfiable—by construction, v_{err} is the only node in $\mathcal{A}$ such that $\nu(v_{\mathsf{err}}) = \mathsf{err}$. If $\psi(v_{\mathsf{err}})$ is unsatisfiable, then $\mathcal{A}$ is safe, and Ufo terminates by declaring the program safe (following Theorem 2.1). Otherwise, Refine is used to compute new labels. In Definition 4.1, we provide a specification of Refine that maintains the soundness of Ufo.

Definition 4.1 (Specification of Refine)**.** If there exists a feasible execution to v_{err} in $\mathcal{A}$, then Refine returns an empty map ($labels = \emptyset$). Otherwise, it returns a map from nodes to labels such that

1. $labels(v_{\mathsf{err}}) \equiv false$,
2. $labels(v_{\mathsf{en}}) \equiv true$, and
3. $\forall (u, v) \in E' \cdot labels(u) \wedge [\![\tau(u, v)]\!] \Rightarrow labels(v)'$, where E' is E restricted to edges along paths to v_{err}.

In other words, the labeling precludes erroneous executions (results in a safe ARG) and maintains well-labeledness of $\mathcal{A}$ (as per Definition 2.2). □

Constructing the ARG ExpandArg adopts a standard *recursive iteration strategy* [Bou93] for unrolling a program's control-flow graph into an ARG. To do so, it makes use of a *weak topological ordering* (WTO) [Bou93] of program locations—see formal definition in Chapter 2. A recursive iteration strategy starts by unrolling the innermost loops until *stabilization*, i.e., until a loop head is covered, before exiting to the outermost loops. We assume that the first location in the WTO is en and the last one is err.

ExpandArg maintains two global maps: AH (active heads) and FN (future nodes). For a loop head l, $\mathsf{AH}(l)$ is the set of nodes $V_\ell \subseteq V$ for location l that are heads of the component being unrolled. When a loop head is covered (line 47), all active heads belonging to its location are removed from AH (line 48). FN maps a location to a single node and is used as a worklist, i.e., it maintains the next node to be explored for a given location. Example 4.1 demonstrates the operation of ExpandArg.

Example 4.1. *Recall our example in Figure 3.2 from Chapter 3. Consider the process of constructing the ARG in Figure 3.2(b) for the program in Figure 3.2(a). First, a WTO for this program is*

$$1\ (2\ 3\ 4\ 5\ 6)\ 7\ 9\ 8.$$

In this example, Post *always returns true. When* ExpandArg *processes node v_2' (i.e., when $v = v_2'$ at line 31),* $\mathsf{AH}(2) = \{v_2\}$*, since the component* (2 3 4 5 6) *representing the loop is being unrolled and v_2 is the only node for location 2 that has been processed. When* Ufo *covers v_2' (line 47), it sets* $\mathsf{AH}(2) = \emptyset$ *(line 48) since the component has stabilized and* Ufo *has to exit it. Here,* $\textsc{WtoExit}(2) = 7$*, so* Ufo *continues processing from node $v_7 = \mathsf{FN}(7)$ (the node for the first location after the loop).*

Suppose Refine returned a new label for node v. When ExpandArg updates $\psi(v)$ (line 42), it *marks* all of its children that do not have labels in *labels*. This is used to strengthen the labels of

[2] The astute reader will probably be able to deduce this fact from the function's name.

v's children with respect to the refined over-approximation of reachable states at v, using the operator Post (line 38). Informally, Refine, typically using DAG interpolants, returns new labels for the ARG that make it well-labeled and safe, but it might not be complete. ExpandArg continues the abstract post computation (AB) from the results of DAG interpolants (IB). Specifically, ExpandArg continues abstract post computation from uncovered nodes in the ARG in order to make the ARG complete—the safe invariant inductive.

ExpandArg only attempts to cover nodes that are loop heads. It does so by checking if the label on a node v is subsumed by the labels on $\mathsf{AH}(\nu(v))$ (line 47). If v is covered, Ufo exits the loop (line 49); otherwise, it adds v to $\mathsf{AH}(\nu(v))$.

Post Operator Ufo is parameterized by the abstract operator, Post. For sound implementations of Ufo, Post should take an edge (u, v) as input and return a formula ϕ such that $\psi(u) \wedge [\![\tau(u, v)]\!] \Rightarrow \phi'$, thus maintaining well-labeledness of the ARG. In the IB case, Post always returns *true*, the weakest possible abstraction. In the combined IB+AB case, Post is driven by an abstract domain, e.g., based on predicate abstraction.

Theorem 4.1 (Soundness)**.** *Given a program P, if a* Ufo *run on P terminates with SAFE, the resulting ARG $\mathcal{A}$ is safe, complete, and well-labeled. If* Ufo *terminates with UNSAFE, then there exists an execution that reaches* err *in P.*

4.2.2 Instantiating Post and Refine

We have presented Ufo without giving a concrete definition of Post and Refine.

For Refine, one possible instantiation is using DAG interpolants, as described in Section 3.4. First, we view the ARG $\mathcal{A}$ as a DAG G and encode its instructions as edge labels $\mathcal{L}_E$. Then, we compute DAG interpolants DItp for the DAG by proving that there are no feasible executions to v_{err}. We assume that Refine returns an empty labeling if no DAG interpolants exist. Note that if no DAG interpolants exist, then DAGCond$(G, \mathcal{L}_E)$ is satisfiable, and we can extract a concrete program execution from en to err from the satisfying assignment.

Let us now explore different instantiations of Post. By varying the implementation of Post, we vary the degree with which AB versus IB drives the construction of a safe inductive invariant.

- In its simplest implementation, Post always returns *true*. Note that this always results in well-labeling of the ARG. In this case, Post is not involved at all in constructing an inductive invariant, and all (useful) labeling of the ARG is performed by DAG interpolants, as computed by the function Refine. Therefore, this is an IB instantiation of Ufo. In fact, this is an implementation of the algorithm we specified in Chapter 3.

- We can implement Post using Cartesian predicate abstraction and instantiate it with some set Preds of predicates. (See Section 2.5 for predicate abstraction definitions.) Specifically,

$$\textsc{Post}(u, v) = \mathsf{CPost}(\psi(u), \tau(u, v)).$$

  In this case, ExpandArg computes an inductive invariant for P—represented as a complete, well-labeled ARG. Then, if the inductive invariant is unsafe, Refine uses DAG interpolants to relabel the ARG such that it is safe and well-labeled. If the result is not a complete ARG, ExpandArg

continues the abstract fixpoint computation from the new labels produced by DAG interpolants. This is hybrid IB/AB technique.

- Similarly, we can implement Post using Boolean predicate abstraction as

$$\textsc{Post}(u, v) = \mathsf{BPost}(\psi(u), \tau(u, v)).$$

This is similar to the Cartesian abstraction instantiation above, and is therefore a hybrid technique as well. The difference is that Boolean abstraction is more precise and more expensive than Cartesian abstraction; we thus consider that this instantiation is driven more by the AB portion of the algorithm. As illustrated in Figure 4.1, the more precise Post is, the more time is spent in ExpandArg, and, therefore, the more AB-like an instantiation is.

- We can also use Ufo as a pure AB technique. Specifically, we use ExpandArg to compute inductive invariants using Boolean or Cartesian abstraction. If the result is an unsafe inductive invariant, we use Refine to find new predicates to add to Preds, and then restart ExpandArg to rebuild the ARG from scratch (i.e., the ARG is reset to a single node—the root), as in *eager abstraction* [BR01].

4.3 Experimental Evaluation

In this section, we describe the implementation and evaluation of different Ufo instantiations.

Implementation The Ufo algorithm is implemented in the UFO_{app} tool, whose architecture, implementation, and optimizations are described in detail in Chapter 7. We mention here some implementation details to provide a clear picture of our experimental setup:

- The UFO_{app} tool is implemented in the popular LLVM compiler infrastructure [LA04]. The verification algorithms operate over LLVM's intermediate representation (bitcode). We use a combination of CIL [NMRW02], `llvm-gcc`, and compiler optimizations supplied by LLVM to transform program written in C into LLVM's intermediate representation.

- For the experiments presented in this chapter, we used MathSAT4 [BCF+08] SMT solver to compute DAG interpolants (by computing sequence interpolants).

- We used the Z3 SMT solver [dMB08] for quantifier elimination required for transforming sequence interpolants to DAG interpolants. Program semantics were encoded using *quantifier-free formulas over linear rational arithmetic* (QF_LRA).

- We implemented independent proof and counterexample checkers to ensure soundness of our results. Our proof checker takes the ARG produced by UFO_{app} when the result is SAFE and checks that it indeed encodes a safe inductive invariant of the program. The counterexample checker unrolls an ARG into a tree and checks each path from v_{en} to v_{err} to see if it is feasible. All results discussed here have been validated by an appropriate checker.

Algorithm	#Solved	#Safe	#Unsafe	#Unsound	Total Time (s)
iUfo	78	22	56	0	8,289
cpUfo	79	22	57	1	7,838
bpUfo	69	17	52	1	11,260
Cp	49	10	39	0	15,363
Bp	71	19	52	1	10,018
Wolverine	38	18	20	5	19,753

Table 4.1: Evaluation of Ufo: results summary.

Evaluation For evaluation, we used the `ntdrivers-simplified`, `ssh-simplified`, and `systemc` benchmarks from the 2012 edition of the Competition on Software Verification (SV-COMP 2012) [Bey12], and the pacemaker benchmarks from [AGC12d]. Overall, we had 105 C programs: 48 safe and 57 buggy. All experiments were conducted on an Intel Xeon 2.66GHz processor running a 64-bit Linux, with a 300 second time and 4GB memory limits per program.

We have evaluated 5 instantiations of Ufo:

1. a pure IB, called iUfo, where Post always returns *true*;
2. a hybrid instantiation with Cartesian predicate abstraction, called cpUfo;
3. a hybrid instantiation with Boolean predicate abstraction, called bpUfo;
4. a pure AB instantiation with Cartesian predicate abstraction, called Cp; and
5. a pure AB instantiation with Boolean predicate abstraction, called Bp.

Recall that Boolean predicate abstraction is more precise, but is exponentially more expensive than Cartesian abstraction.

The results are summarized in Table 5.1. For each configuration, we show the number of instances solved (#Solved), number of safe (#Safe) and unsafe (#Unsafe) instances solved, number of unsound results (#Unsound), where a result is unsound if it does not agree with the benchmark categorization in [Bey12], and the total time.

On these benchmarks, cpUfo outperforms other configurations, both in total time and number of instances solved. The iUfo configuration is a very close second. We have also compared our results against the IB tool Wolverine [KW11] that implements a version of Impact [McM06] algorithm. All configurations of Ufo perform significantly better than Wolverine.

Furthermore, we compared our tool against the results of the extensive study reported in [BK11] for the state-of-the-art AB tools CPAchecker [BK11], Blast [BHJM07], and SatAbs [CKSY05]. Both iUfo and cpUfo configurations are able to solve all buggy `transmitter` examples. However, according to [BK11], CPAchecker, Blast, and SatAbs are unable to solve most of these examples, even though they are run on a faster processor with a 900s time limit and 16GB of memory. Additionally, on the `ntdrivers-simplified`, iUfo, cpUfo, and bpUfo perform significantly better than all of the aforementioned tools.

Table 4.2 presents a detailed comparison between different instantiations of Ufo on 32 (out of 105) programs. In the table, we show time, number of iterations of the main loop in UfoMain (#Iter), and time spent in interpolation (#iTime) and post (#pTime), respectively. Times taken by other parts of

the algorithm (such as Clean) were insignificant and are omitted. The Cp configuration was not able to solve all but one of these examples, and is omitted as well.

In this sample, cpUfo is best overall, however, it is often not the fastest approach on any given example. This is representative of its performance over the whole benchmark. As expected, both iUfo and cpUfo spend most of their time in computing interpolants (IB), while bpUfo and Bp spend most of their time in predicate abstraction (AB).

The results demonstrate a synergy between IB and AB parts of the analysis. For example, in `toy1_BUG` and `s3_srvr_1a`, predicate abstraction decreases the number of required iterations. Several of the buggy examples from the `token_ring` family cannot be solved by the pure IB iUfo configuration alone. However, there are also some undesired interactions. For many of the safe cases that require a few iterations, iUfo performs better than other combinations. For many unsafe cases that bpUfo can solve, it performs much better alone than in a combination.

In summary, our results show that the novel IB, DAG-interpolation-based algorithm that underlies Ufo (iUfo configuration) is very effective compared to the state-of-the-art approaches. Furthermore, our results suggest potential advantages in combining IB and AB approaches, with cpUfo performing the best overall. However, there are also some interactions where the combination does not result in the best of the individual approaches. Inspired by these results, in Chapter 5 we explore deeper and more general combinations of AB and IB techniques.

4.4 Related Work

In this section, we place Ufo in the context of related work. Specifically, we compare it with the most related IB and AB techniques.

Interpolation-based Verification In its IB instantiation, Ufo is a novel interpolation-based verification algorithm: It extends McMillan's original IB technique, LAWI [McM06], by unrolling the program into a DAG instead of a tree and by using DAG interpolants discharge all infeasible unsafe executions and to compute new ARG labels. In effect, Ufo uses the SMT solver to enumerate acyclic program paths, whereas LAWI enumerates those paths explicitly via a tree unrolling. Furthermore, when instantiated with a predicate abstract domain, Ufo extends LAWI by using an abstract post operator to *push* labels computed by DAG interpolants down the abstract reachability graph. As we show in our experiments, this can lead to fewer iterations and faster verification.

In some sense, our IB instantiation of Ufo is similar to McMillan's first finite-state model checking algorithm with interpolants [McM03]. McMillan's algorithm unrolls a symbolic transition relation a finite number of times. If bounded model checking does not find a counterexample for the bounded unrolling, an invariant is hypothesized using interpolants. In Ufo, we unroll the program's transition relation at a fine-grained level, unrolling loops instead of the whole relation; and compute fine-grained invariants, per program location instead of global (monolithic) invariants for the whole program.

Synergy [GHK+06] and Dash [BNRS08] can also be viewed as interpolation-based algorithms, in the sense that they do not utilize an abstract domain and abstract fixpoint computation. The difference is they use weakest preconditions (WP) over infeasible program executions to hypothesize a safe inductive invariant. In contrast, Ufo examines (refines) multiple program paths at the same time. Moreover, Ufo uses interpolants for refinement, an approach that has been shown to provide more relevant predicates

than WP-based refinement [HJMM04]. We believe that our multi-path refinement strategy can be easily implemented in Dash to generate test-cases for multiple frontiers or split different regions at the same time.

Abstraction-based Verification *Lazy abstraction* [HJMS02] is the closest AB algorithm to Ufo. Lazy abstraction operates by unrolling the program into an abstract reachability tree and labeling nodes using a predicate abstract domain. If a path through the tree reaches an unsafe error node, new predicates are computed and the subtree that reached the unsafe node is rebuilt using the refined abstract domain—in comparison, *eager* abstraction rebuilds the whole tree with the new set of predicates. Ufo can be seen as extending lazy abstraction in two directions. First, Ufo unrolls a program into a DAG instead of a tree, providing the same advantages as compared to LAWI. Second, it uses interpolants to directly label the ARG, and only applies predicate abstraction to the *frontier* nodes that are not known to reach an error location.

Slam is one of the first software verification algorithms to marry predicate abstraction and CEGAR. Its success was primarily due to its application to Windows device driver verification and bug finding. Specifically, SLAM showed that a small set of predicates is often sufficient for proving API-usage properties, e.g., locking and unlocking patterns, of device drivers. SLAM implements an eager abstraction refinement loop: First, an invariant is computed using Boolean (or Cartesian) predicate abstraction starting from an empty set of predicates. A refinement phase detects abstract counterexamples and checks one of them. If the counterexample is spurious, new predicates are added to the predicate set, and the abstract fixpoint is restarted using a refined predicate abstract domain. Our AB instantiation of Ufo in Section 4.3 is similar to Slam in the sense that is eager: resets the ARG when an unsafe inductive invariant is found. However, our instantiation uses DAG interpolants to find predicates for potentially exponentially many abstract counterexamples, instead of a single abstract counterexample.

We are not the first to apply interpolation to multiple program paths. In [EKS06], Esparza et al. use interpolants to find predicates that eliminate multiple spurious counterexamples simultaneously. Their algorithm uses an eager abstraction-refinement loop and a BDD-based interpolation procedure. In contrast, the refinement in Ufo uses an SMT-solver-based interpolation procedure, providing more efficient implementations and more expressivity in terms of logics used to model program semantics.

4.5 Conclusion

In this chapter, we presented Ufo, an algorithm that combines abstraction-based and interpolation-based software verification techniques. Traditional static analyses for verification fall under the AB category: they employ an abstract domain (implicitly or explicitly) and an abstract fixpoint computation to produce an inductive program invariant. AB techniques spend most of their time executing the program under abstract semantics (using abstract transformers), which is typically an expensive process for expressive abstract domains. IB techniques are a new class of techniques that eschews use of expensive abstract transformers; instead, IB techniques examine program executions using efficient theorem provers and hypothesize safe inductive invariants using Craig interpolants.

Ufo is parameterized by the degree with which IB and AB techniques drive the verification process. On the one hand, it can be instantiated as a pure IB technique: an implementation of the DAG-interpolation-based technique presented in Chapter 3. On the other hand, it can be instantiated as an

AB technique using a predicate abstract domain, where DAG interpolants are simply used to refine the abstract domain by augmenting the predicate set. In the middle, it is a hybrid IB/AB technique, where IB and AB verification alternate, reusing the results of each other.

We have evaluated different instantiations of Ufo: pure IB, pure AB (with Boolean and Cartesian abstraction), and hybrid IB/AB. Our evaluation of these different instantiations on a suite of C programs demonstrates (1) the power of our DAG-interpolation-based algorithm in relation to other IB techniques and (2) the advantages gained from combining AB and IB techniques. In Chapter 5, we extend Ufo to utilize infinite-height domains for the AB portion, as opposed to just predicate abstraction, and show how DAG interpolants can be used to refine results of abstract interpretation.

Program	iUfo			cpUfo				bpUfo				Bp			
	Time	Iter	iTime	Time	#Iter	iTime	pTime	Time	Iter	iTime	pTime	Time	Iter	iTime	pTime
Unsafe Programs															
kundu1	-	-	-	24.22	4	20.3	1.84	122.88	4	56.9	54.66	33.39	3	20.23	10.95
kundu2	1.24	2	1.16	2.74	2	2.08	0.6	8.15	2	1.2	5.66	8.6	2	3.49	4.3
s3_srvr_11	1.91	4	1.67	2.78	4	1.58	0.89	118.41	4	1.72	112.76	4.25	3	2.6	1.37
s3_srvr_12	4.17	4	3.85	5.07	3	3.44	1.36	5.36	3	3.58	1.44	8.19	3	5.85	1.91
token_ring.08	12.34	4	11.84	13.5	4	11.07	1.91	19.64	3	3.7	14.62	14.15	3	1.85	11.12
token_ring.09	12.54	4	11.98	22.66	4	19.72	2.35	-	-	-	-	167.49	3	3.85	157.58
token_ring.10	15.6	4	15.05	14.02	3	11.99	1.69	-	-	-	-	-	-	-	-
token_ring.11	29.69	4	29.08	22.47	4	18.52	3.19	156.76	3	4.57	145.99	66.59	3	4.21	58.68
token_ring.12	26.94	4	26.31	13.98	3	11.45	2.15	-	-	-	-	-	-	-	-
token_ring.13	36.56	4	35.76	34.17	4	29.38	4.02	-	-	-	-	-	-	-	-
token_ring.14	10.3	3	9.99	33.49	4	29.17	3.59	-	-	-	-	-	-	-	-
token_ring.15	51.79	4	51.11	34.19	4	29.18	4.17	-	-	-	-	-	-	-	-
toy1	96.49	10	89.08	79	9	68.04	6.98	13.54	3	4.77	7.96	-	-	-	-
toy2	12.83	5	12.24	60.73	8	50.71	6.14	-	-	-	-	-	-	-	-
ddd3	0.66	4	0.5	0.19	2	0.04	0.11	0.18	2	0.03	0.11	0.27	2	0.05	0.2
Safe Programs															
pc_sfifo_1	-	-	-	-	-	-	-	3.51	3	2.24	0.79	-	-	-	-
s3_clnt_1	11.03	10	8.18	15.68	10	8.2	4.5	-	-	-	-	14.52	5	1.92	8.21
s3_clnt_2	16	10	11.35	20.02	10	10.86	4.67	-	-	-	-	-	-	-	-
s3_clnt_3org	28.87	11	17.35	37.08	11	17.6	8.17	-	-	-	-	-	-	-	-
s3_clnt_3	13.02	10	9.14	17.42	10	9.01	4.45	-	-	-	-	-	-	-	-
s3_clnt_4	13.4	10	9.62	17.45	10	9.16	4.58	-	-	-	-	-	-	-	-
s3_srvr_1a	5.2	10	2.95	5.16	8	2.32	1.07	0.76	4	0.17	0.39	0.43	3	0.07	0.26
s3_srvr_1b	1.37	7	1	2.9	7	1.71	0.69	0.89	5	0.47	0.28	-	-	-	-
s3_srvr_2	171.15	17	116.82	184.01	17	112.65	18.65	-	-	-	-	-	-	-	-
s3_srvr_3	133.07	17	99.96	147.55	17	98.69	16.02	-	-	-	-	33.71	5	1.07	21.18
s3_srvr_4	-	-	-	-	-	-	-	-	-	-	-	8	4	0.74	5.36
s3_srvr_8	101.4	14	76.6	115.08	14	73.9	17.62	-	-	-	-	-	-	-	-
token_ring.01	98.18	18	81.58	23.64	10	17.72	1.78	0.69	4	0.27	0.23	0.69	4	0.19	0.31
token_ring.02	-	-	-	-	-	-	-	2.15	4	0.71	0.7	2.63	4	1.06	0.59
token_ring.03	-	-	-	-	-	-	-	76.18	4	4.74	37.66	-	-	-	-
token_ring.04	-	-	-	-	-	-	-	-	-	-	-	152.62	4	10.82	2.45
token_ring.05	-	-	-	-	-	-	-	-	-	-	-	149.35	4	8.25	97.48

Table 4.2: Evaluation of Ufo: detailed results.

Chapter 5

Abstract Interpretation and Interpolation-based Verification

"There is no abstract art. You must always start with something. Afterward you can remove all traces of reality."

— Pablo Picasso

5.1 Introduction

As we have discussed in Chapter 1, abstraction-based verification relies on an abstract domain D to iteratively compute an inductive invariant of a given program. This process is typically described as an abstract interpretation (AI) of program semantics. The price of AI's efficiency is false alarms (i.e., inability to find a safe I) that are introduced through imprecision inherent in many steps of the analysis, e.g., widening, join, inexpressiveness of the domain, etc.

In this chapter, we describe Vinta[1], an iterative algorithm that uses interpolants to refine and guide AI away from false alarms. Vinta marries the efficiency of AI with the precision of Bounded Model Checking (BMC) [BCCZ99] and the ability to generalize from concrete executions of interpolation-based software verification. Vinta can be viewed through two different lenses:

- Vinta is an improved and generalized version of Ufo (described in Chapter 4), where arbitrary abstract domains can be used for the AB portion of the algorithm. Ufo is restricted to predicate abstraction, a finite-height domain: with a given set of predicates, only a finite number of invariants is expressible (as Boolean combinations of predicates). Vinta extends Ufo to general abstract domains, like intervals and octagons, by tackling an array of issues including widening and abstraction. The array of possible instantiations of Vinta strictly subsume those of Ufo.
- Vinta is a refinement loop for invariant generation with AI. Specifically, Vinta computes an inductive invariant using some abstract domain; if the result is an unsafe inductive invariant, Vinta uses an extension of DAG interpolants to recover imprecision lost due to widening, joins, or even inexpressiveness in the abstract domain.

[1] Verification with INTerpolation and Abstract interpretation.

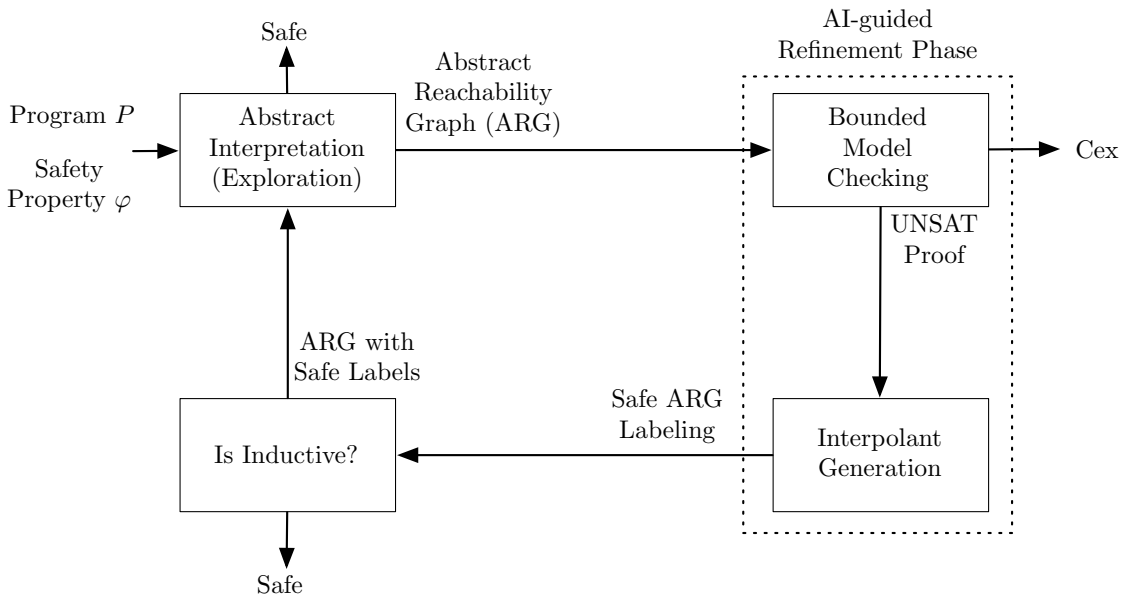


Figure 5.1: High level description of Vinta.

Overview The main phases of the algorithm are shown in Figure 5.1. Given a program P and a safety property φ, Vinta starts by computing an inductive invariant I of P using an abstract domain D (the AI phase). If I is a safe inductive invariant, then P is safe as well. Otherwise, Vinta goes to a novel refinement phase. First, refinement uses BMC to check for a counterexample in the explored part of P. Second, if BMC fails to find a counterexample, it uses an interpolation-based procedure to strengthen I to I'. If I' is not inductive (checked in the "Is Inductive?" phase), the AI phase is repeated to weaken I' to include all reachable states of P. This process continues until either a safe inductive invariant or a counterexample is found, or resources (i.e., time or memory) are exhausted.

Our presentation of Vinta closely follows that of Ufo: The BMC and interpolant generation phases are used to compute a new form of DAG interpolants, called *Restricted DAG Interpolants*, that are used to recover imprecision in the AI phase.

Contributions

We summarize this chapter's contributions as follows:

- We present an extension of the algorithm presented in Chapter 4 to arbitrary abstract domains (instead of predicate abstraction) for the AB portion. The resulting algorithm is a refinement loop for invariant generation with abstract interpretation.

- We present the notion of restricted DAG interpolants, which are DAG interpolants that utilize invariants computed via abstract domains to "guide" the interpolation process.

- We evaluate instantiations of our algorithm and show that it can outperform state-of-the-art tools from the literature as well as previous instantiations from Chapter 4.

Organization

This chapter is organized as follows:

- In Section 5.2, we illustrate the operation of Vinta on a simple example.
- In Section 5.4, we formally present Vinta along with its widening and refinement strategies.
- In Section 5.5, we present a thorough evaluation of Vinta with different abstract domains and refinement strategies.
- In Section 5.6, we describe closely related work.
- Finally, in section 5.7, we conclude the chapter.

5.2 Illustrative Example

In this section, we illustrate the operation of Vinta for proving safety of a program P from [GCNR08], shown in Figure 5.2(a). P is known to be hard to analyze without refinement, and even the refinement approaches of Gulavani et al. [GHK+06] and Wang et al. [WYGI07] fail to solve it (see Gulavani et al. [GCNR08] for details). Dagger [GCNR08] (the state-of-the-art in AI refinement) solves it using the domain of polyhedra by computing the safe inductive invariant $x \leqslant y \leqslant 100x$. Here, we show how Vinta solves the problem using the Box (intervals) domain and refinement to compute an alternative safe inductive invariant:

$$x \geqslant 4 \Rightarrow y > 100.$$

In this example, the refinement must recover imprecision lost due to widening and join, and extend the base-domain with disjunction. All of this is done automatically using a new form of DAG interpolants.

Step 1.1: AI Vinta works on a *cutpoint graph* (CPG) of a program: a collapsed CFG where the only nodes are cutpoints (loop heads), entry, and error locations. A CPG for P is shown in Figure 5.2(b).

Vinta uses a typical AI computation following the *recursive iteration strategy* [Bou93] and widening at every loop unrolling. Additionally, it records the finite traces explored by AI in an *Abstract Reachability Graph* (ARG). Analogous to Ufo in Chapter 4, an ARG is an unrolling of the CPG. Each node u of an ARG corresponds to some node v of a CPG, and is labeled with an over-approximation of the set of states reachable at that point.

Figure 5.2(c) shows the ARG from the first AI computation on P. Each node v_i in the ARG refers to node ℓ_i in the CPG. The superscript in nodes v_2^a, v_2^b, v_2^c, and v_2^d is used to distinguish between the different unrollings of the loop at ℓ_2. The labels of the nodes v_2^a, v_2^b, and v_2^c over-approximate the states reachable before the first, second, and third iterations of the loop, respectively. The node v_2^c is said to be covered (i.e., subsumed) by $\{v_2^a, v_2^b\}$. The labels of the set $\{v_2^a, v_2^b\}$ form an inductive invariant $\mathcal{I}_1 \equiv (x \geqslant 0 \wedge y \geqslant 0)$. The node v_2^d is called an *unexplored child*, and has no label and no children. It is used later when AI computation is restarted. Finally, note that $\mathcal{I}_1$ is not safe (the error location v_e is not labeled by *false*), and thus refinement is needed.

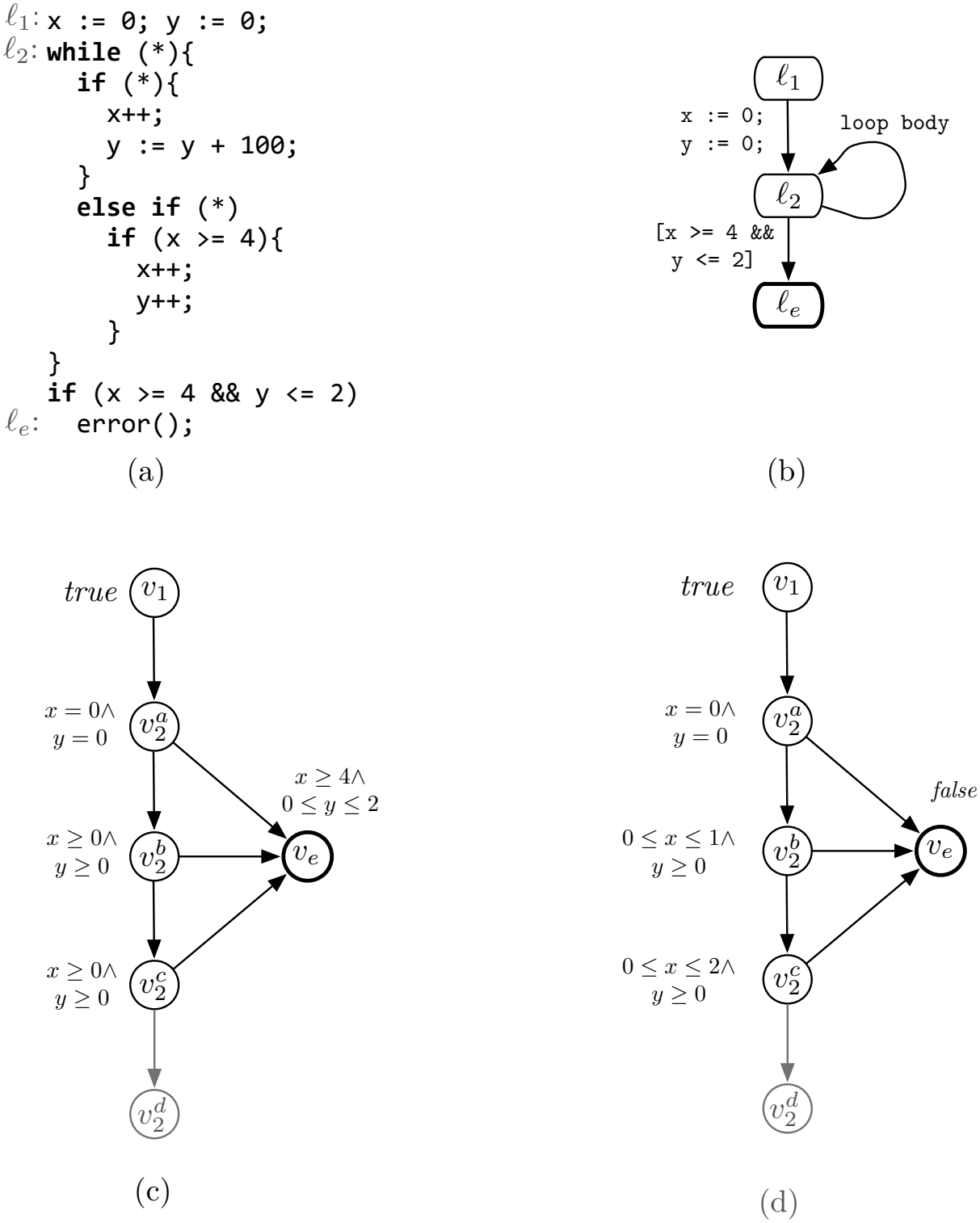


Figure 5.2: Illustration of Vinta on a safe program.

Step 1.2: AI-guided Refinement First, Vinta uses a BMC-style technique [GCS11] to check, using an SMT solver, whether the current ARG has a feasible execution to the error node ℓ_e. There is no such

execution in our example (see Figure 5.2(c)) and the algorithm moves to the next phase.

The second phase of refinement is based on an extension of DAG interpolants (Chapter 3) that we call *Restricted DAG Interpolants*. Specifically, the procedure takes the current ARG (Figure 5.2(c)) and its labeling and, using restricted DAG interpolants, produces a new safe (but not necessarily inductive) labeling shown in Figure 5.2(d). From an abstract interpretation perspective, refinement reversed the effects of widening by restoring the upper bounds on x. Note that the new labels are stronger than the original ones—this is guaranteed by the procedure and the original labels are used to guide it. Alternatively, from an interpolation-based verification perspective, we used restricted DAG interpolants to compute a safe well-labeling of the ARG that took into account the labeling produced by the Box abstract domain. Intuitively, restricted DAG interpolants answer the question: *what is needed to strengthen the current labels of the ARG in order to make it safe and well-labeled?* For instance, interpolants strengthened the label of v_2^b by simply adding the constraint $x \leqslant 1$ to the AI label, instead of computing a completely new label. We say the refinement here is *guided* by AI.

Step 1.3: Is Inductive? The new ARG labeling (Figure 5.2(d)) is not inductive since the label of v_2^c is not contained in the label of v_2^b (checked by an SMT solver), and another AI phase is started.

Step 2.1: AI (again) AI is restarted "lazily" from the nodes that have unexplored children. Here, v_2^c is the only such node. This ensures that AI is restarted from the inner-most loop where the invariant is no longer inductive. First, the label of v_2^c is converted into an element of an abstract domain by a given abstraction function. In our example, the label is immediately expressible in Box, so this step is trivial. Then, AI computation is restarted as usual.

In the following four iterations (omitted here), refinement works with the AI-based exploration to construct a safe inductive invariant $x \geqslant 4 \Rightarrow y > 100$. Note that since the invariant contains a disjunction, this means refinement had to recover from imprecision of join (as well as recovering from imprecision due to widening shown above).

This example is simple enough to be solved with other interpolation-based techniques, but they require more iterations. The Ufo algorithm (Chapter 4), without AI-guided exploration and refinement, needs nine iterations, and a version of Vinta with unguided refinement from Ufo needs seven. Our experiments suggest that this translates into a significant performance difference on bigger programs.

5.3 Preliminaries: Abstract Domains

Abstract and concrete domains are often presented as Galois-connected lattices [CC77]. In this chapter, we use a more operational presentation. Without loss of generality, we restrict the concrete domain to a set B of all Boolean expressions over program variables (as opposed to the powerset of concrete program states). We define an abstract domain as a tuple $\mathcal{D} = (D, \top, \bot, \sqcup, \triangledown, \alpha, \gamma)$, where

- D is the set of abstract elements with two designated elements;
- $\top, \bot \in D$, called *top* and *bottom*, respectively;
- two binary functions $\sqcup, \triangledown : D \times D \to D$, called *join* and *widen*, respectively; and
- two functions: an *abstraction* function $\alpha : B \to D$ and a *concretization* function $\gamma : D \to B$.

Algorithm 3 The Vinta algorithm.

```
1:  function VintaMain(Program P)
2:      create nodes v_en, v_err
3:      ψ(v_en) ← true
4:      ν(v_en) ← en
5:      ψ(v_err) ← false
6:      ν(v_err) ← err
7:      marked(v_en) ← true
8:      labels ← ∅
9:      while true do
10:         ExpandArg()
11:         if ψ(v_err) is unsatisfiable then
12:             return SAFE
13:         labels ← Refine(A)
14:         if labels = ∅ then
15:             return UNSAFE

16: function GetFutureNode(ℓ ∈ L)
17:     if FN(ℓ) is defined then
18:         return FN(ℓ)
19:     create node v
20:     ψ(v) ← true
21:     ν(v) ← ℓ
22:     FN(l) ← v
23:     return v

24: function ExpandNode(v ∈ V)
25:     if v has children then
26:         for all (v, w) ∈ E do
27:             FN(ν(w)) ← w
28:     else
29:         for all (ν(v), T, ℓ) ∈ δ do
30:             w ← GetFutureNode(ℓ)
31:             E ← E ∪ {(v, w)}
32:             τ(v, w) ← T
```

The functions respect the expected properties: $\alpha(\mathit{true}) = \top$, $\gamma(\bot) = \mathit{false}$, for $x, y, z \in D$, if $z = x \sqcup y$ then $\gamma(x) \vee \gamma(y) \Rightarrow \gamma(z)$, etc. Note that D has no meet and no abstract order—we do not use them. Finally, we assume that for every program statement T, there is a sound abstract transformer $\mathsf{APost}_{\mathcal{D}}$ such that if $d_2 = \mathsf{APost}_{\mathcal{D}}(T, d_1)$ then $\gamma(d_1) \wedge [\![T]\!] \Rightarrow \gamma(d_2)'$, where $d_1, d_2 \in D$, and for a formula X, X' is X with all variables primed.

5.4 The Vinta Algorithm

In this section, we formally describe Vinta and discuss its properties. Vinta is shown in Algorithms 3 and 4. Vinta is based on Ufo, but improves it in several directions:

1. It extends Ufo to arbitrary abstract domains using a new form of widening;
2. While in theory Vinta is compatible with the refinement strategy of Ufo, in Section 5.4.3 we describe the shortcomings of Ufo's refinement in our setting and present a new and advanced refinement strategy.
3. It employs a more efficient covering strategy (line 53): instead of checking subsumption against nodes of the current unrolling of a given loop—as in Ufo—Vinta checks subsumption against all visited nodes in the construction of an ARG.

The following presentation of Vinta closely follows that of Ufo in Chapter 4; we point out and explain the major differences: widening, refinement, abstract post computation, and covering.

5.4.1 Main Algorithm

VintaMain Function VintaMain in Algorithm 3 implements the loop in Figure 5.1. It takes a program $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ and checks whether the error location err is reachable. Without loss

Algorithm 4 Vinta's ExpandArg algorithm.

```
33: function ExpandArg
34:     vis ← ∅
35:     FN ← ∅
36:     FN(err) ← v_err
37:     v ← v_en
38:     while true do
39:         ℓ ← ν(v)
40:         ExpandNode(v)
41:         if marked(v) then
42:             marked(v) ← false
43:             ψ(v) ← ComputePost(v)
44:             ψ(v) ← WidenWith({ψ(u) | u ∈ vis(ℓ)}, ψ(v))
45:             for all (v, w) ∈ E do
46:                 marked(w) ← true
47:         else if labels(v) is defined then
48:             ψ(v) ← labels(v)
49:             for all {(v, w) ∈ E | labels(w) is undefined} do
50:                 marked(w) ← true
51:         vis(ℓ) ← vis(ℓ) ∪ {v}
52:         if v = v_err then break
53:         if Smt.IsValid(ψ(v) ⇒ ⋁_{u∈vis(ℓ),u≠v} ψ(u)) then
54:             erase FN(ℓ)
55:             repeat
56:                 ℓ ← WtoExit(ℓ)
57:             until FN(ℓ) is defined
58:             v ← FN(ℓ)
59:             erase FN(ℓ)
60:             for all {(v, w) ∈ E | ∄u ≠ v · (u, w) ∈ E} do
61:                 erase FN(ν(w))
62:         else
63:             ℓ ← WtoNext(ℓ)
64:             v ← FN(ℓ)
65:             erase FN(ℓ)
```

of generality, we assume that every location in $\mathcal{L}$ is reachable from en and can reach err (ignoring the semantics of actions). In addition, we assume that all nodes in $\mathcal{L}$ are cutpoints (loop heads), and every action is a loop-free program segment between two cutpoints (as in *large-block encoding* [BCG+09]). We call the induced CFG a *cutpoint graph* (CPG). VintaMain maintains a globally accessible ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$. If VintaMain returns SAFE, then $\mathcal{A}$ is safe, complete, and well-labeled (thus proving safety of P by Theorem 2.1).

VintaMain is parameterized by (1) the abstract domain $\mathcal{D}$ and (2) the refinement function Refine. First, an ARG is constructed by ExpandArg using an abstract transformer $\mathsf{APost}_{\mathcal{D}}$. For simplicity of presentation, we assume that all labels are Boolean expressions that are implicitly converted to and from $\mathcal{D}$ using functions α and γ, respectively. ExpandArg always returns a complete and well-labeled ARG. So, on line 11, VintaMain only needs to check whether the current ARG is safe. If the check fails, Refine is called to find a counterexample and remove false alarms. We describe our implementation of Refine in Section 5.4.3, but the correctness of the algorithm depends only on the following abstract specification of Refine, as introduced in Chapter 4.

Definition 5.1 (Specification of Refine (Chapter 4))**.** Refine returns an empty map ($labels = \emptyset$) if there exists a feasible execution from v_{en} to v_{err} in $\mathcal{A}$. Otherwise, it returns a map *labels* from nodes to Boolean expressions such that

1. $labels(v_{\mathsf{en}}) \equiv true$,
2. $labels(v_{\mathsf{err}}) \equiv false$,
3. $\forall (u, v) \in E \cdot labels(u) \wedge [\![\tau(u, v)]\!] \Rightarrow labels(v)'$. □

In our case, refinement uses BMC and interpolation through an SMT solver to compute labels, therefore, if no labels are found, refinement produces a counterexample as a side effect.

Whenever Refine returns a non-empty labeling (i.e., false alarms were removed), VintaMain calls ExpandArg again. ExpandArg uses *labels* to relabel the existing ARG nodes and uses $\mathsf{APost}_{\mathcal{D}}$ to expand the ARG further (if the resulting labeling is not an inductive invariant).

The ExpandArg Algorithm ExpandArg constructs the ARG in a *recursive iteration strategy* [Bou93], It assumes existence of a *weak topological ordering* (WTO) [Bou93] of the CPG and two functions, WtoNext and WtoExit, as described in Chapter 2.

ExpandArg maintains two local maps: vis and FN. vis maps a cutpoint ℓ to the set of visited nodes corresponding to ℓ, and FN maps a cutpoint ℓ to the first unexplored node $v \in V$ such that $\nu(v) = \ell$. The predicate *marked* specifies whether a node is labeled using AI (*marked* is *true*) or it gets a label from the map *labels* produced by Refine (*marked* is *false*). Marks are propagated from a node to children (lines 45 and 49). Initially, the entry node is marked (line 7), which causes all of its descendants to be marked as well. AI over all incoming edges of a node v is done using ComputePost(v) that over-approximates $\mathrm{Post}_{\mathcal{D}}$ computations over all predecessors of a node v (that are in vis).

Note that Vinta uses an ARG as an efficient representation of a disjunctive invariant: for each cutpoint $\ell \in \mathcal{L}$, the disjunction $\bigvee_{v \in \mathsf{vis}(\ell)} \psi(v)$ is an inductive invariant. The key to efficiency is two-fold. First, a possibly expensive abstract subsumption check is replaced by an SMT check (line 53). Second, inspired by [GCNR08], an expensive powerset widening is replaced by a simple widening scheme, WidenWith, that lifts base domain widening $\triangledown$ to a widening between a set and a *single* abstract element. We describe WidenWith in detail in Section 5.4.2.

Abstract Post The function ComputePost propagates and joins labels (abstract states) to some node v. Formally:

$$\textsc{ComputePost}(v) = \bigsqcup \{\mathsf{APost}_{\mathcal{D}}(\tau(u,v), \alpha(\mathit{labels}(u))) \mid (u,v) \in E, u \in \mathsf{vis}\}\,.$$

In other words, abstract post under domain $\mathcal{D}$ is computed along each edge ending in v, and all of the resulting abstract states are joined.

5.4.2 Widening

In this section, we describe the powerset widening operator WidenWith used by Vinta.

Definition 5.2 (Specification of WidenWith)**.** Let $\mathcal{D} = (D, \top, \bot, \sqcup, \triangledown, \alpha, \gamma)$ be an abstract domain. An operator $\triangledown_W : \mathcal{P}_f(D) \times D \to D$ is a WidenWith operator if and only if it satisfies the following two conditions:

1. (soundness) for any finite set $X \subseteq D$ and $y \in D$, $(\gamma(X) \vee \gamma(y)) \Rightarrow (\gamma(X) \vee \gamma(X \triangledown_W y))$;
2. (termination) for any finite set $X \subseteq D$ and a sequence $\{y_i\}_i \in D$, the sequence $\{Z_i\}_i \subseteq D$, where $Z_0 = X$ and $Z_i = Z_{i-1} \cup \{Z_{i-1} \triangledown_W y_i\}$, converges, i.e., $\exists i \cdot \gamma(Z_{i+1}) \Rightarrow \gamma(Z_i)$,

where $\gamma(X) \equiv \bigvee_{x \in X} \gamma(x)$, for some set of abstract elements X. □

Note that unlike traditional powerset widening operators (e.g., Bagnara et al. [BHZ06]), WidenWith is defined for a pair of a set and an element (and not a pair of sets). It is inspired by the widening operator $\triangledown_T^p$ of Gulavani et al. [GCNR08], but differs from it in three important aspects.

1. We do not require that if $z = \textsc{WidenWith}(X, y)$, then z is "bigger" than y, i.e., $\gamma(y) \Rightarrow \gamma(z)$. Intuitively, if X and y approximate sets of reachable states, then z over-approximates the *frontier* of y (i.e., states in y but not in X).
2. Our termination condition is based on concrete implication (and not on an abstract order).
3. We do not require that X or the sets $\{Z_i\}_i$ in Definition 5.2 contain only "maximal" elements [GCNR08].

These differences give us more freedom in designing the operator and significantly simplify the implementation.

We now describe two implementations of WidenWith: the first, $\textsc{WidenWith}_{\sqcup}$, is based on $\triangledown_T^p$ from [GCNR08] and applies to any abstract domain; the second, $\textsc{WidenWith}_{\vee}$, requires an abstract domain that supports disjunction ($\vee$), i.e., precise join, and set difference ($\setminus$). One example of such a domain is Boxes [GC10]. The operators are defined as follows:

$$\textsc{WidenWith}_{\sqcup}(\emptyset, y) = y \tag{5.1}$$

$$\textsc{WidenWith}_{\vee}(\emptyset, y) = y \tag{5.2}$$

$$\textsc{WidenWith}_{\sqcup}(X, y) = x \triangledown (x \sqcup y) \tag{5.3}$$

$$\textsc{WidenWith}_{\vee}(X, y) = \Big((\bigvee X) \triangledown (\bigvee X \vee y)\Big) \setminus \bigvee X \tag{5.4}$$

where $x \in X$ is picked non-deterministically from X.

Theorem 5.1 ($\textsc{WidenWith}_{\{\vee,\sqcup\}}$ Correctness)**.** $\textsc{WidenWith}_{\sqcup}$ *and* $\textsc{WidenWith}_{\vee}$ *satisfy the two conditions of Definition 5.2.*

Algorithm 5 Ufo's refinement technique.

1: **function** UfoRef(ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$)
2: $\quad\mathcal{L}_E \leftarrow$ EncodeBmc($\mathcal{A}$)
3: $\quad$DItp $\leftarrow$ ComputeDItp($(V, E, v_{\mathsf{en}}, v_{\mathsf{err}}), \mathcal{L}_E$)
4: $\quad$**return** DecodeBmc(DItp)

5.4.3 Refinement

In this section, we formalize Vinta's refinement strategy. We start by describing *Restricted DAG Interpolants* (RDI): an extension of a DAG Interpolants that utilizes information from abstract interpretation to guide the process of computing interpolants.

In the rest of this section, we write

- F for a set of formulas;
- $G = (V, E, v^{en}, v^{ex})$ for a DAG with an entry node $v^{en} \in V$ and an exit node $v^{ex} \in V$, where v^{en} has no predecessors, v^{ex} has no successors, and every node $v \in V$ lies on a (v^{en}, v^{ex})-path;
- $desc(v)$ and $anc(v)$ for the sets of edges that can reach and are reachable from a node $v \in V$, respectively;
- $\mathcal{L}_E : E \to F$ and $\mathcal{L}_V : V \to F$ for maps from edges and vertices to formulas, respectively; and
- $FV(\varphi)$ for the set of free variables in a given formula φ.

Definition 5.3 (Restricted DAG Interpolant (RDI))**.** Let G, $\mathcal{L}_E$, and $\mathcal{L}_V$ be as defined above. An RDI for $G, \mathcal{L}_E$, and $\mathcal{L}_V$ is a map $\text{RDItp} : V \to F$ such that

1. $\forall e = (v_i, v_j) \in E \cdot \big(\text{RDItp}(v_i) \wedge \mathcal{L}_V(v_i) \wedge \mathcal{L}_E(e)\big) \implies \text{RDItp}(v_j) \wedge \mathcal{L}_V(v_j)$,
2. $\text{RDItp}(v^{en}) \equiv \mathit{true}$,
3. $\big(\text{RDItp}(v^{ex}) \wedge \mathcal{L}_V(v^{ex})\big) \equiv \mathit{false}$, and
4. $\forall v_i \in V \cdot FV(\text{RDItp}(v_i)) \subseteq \Big(\bigcup_{e \in desc(v_i)} FV(\mathcal{L}_E(e))\Big) \cap \Big(\bigcup_{e \in anc(v_i)} FV(\mathcal{L}_E(e))\Big)$. □

Whenever $\forall v \cdot \mathcal{L}_V(v) \equiv \mathit{true}$, we say that an RDI is *unrestricted* or simply a DAG Interpolant (DI). Intuitively, when all node labels are set to *true*, the definition of RDI reduces to that of DI, as point 1 of Definition 5.3 simplifies to

$$\forall e = (v_i, v_j) \in E \cdot \big(\mathcal{L}_V(v_i) \wedge \mathcal{L}_E(e)\big) \implies \mathcal{L}_V(v_j).$$

In general, in a proper RDI (i.e., when $\exists v \cdot \mathcal{L}_V(v) \neq \mathit{true}$), $\text{RDItp}(v)$ is not an interpolant by itself, but is a projection of an interpolant to $\mathcal{L}_V(v)$. That is, $\text{RDItp}(v)$ is the restriction needed to turn $\mathcal{L}_V(v)$ into an interpolant. Thus, an RDI can be weaker (and possibly easier to compute) than a DI.

Ufo Refinement Ufo's refinement procedure is shown in Algorithm 5, where the function EncodeBmc encodes the edge labeling $\mathcal{L}_E$ of the ARG, the function ComputeDItp computes DAG interpolants, and the function DecodeBmc removes subscripts (introduced by encoding) from DItp to produce a well-labeling of the ARG.

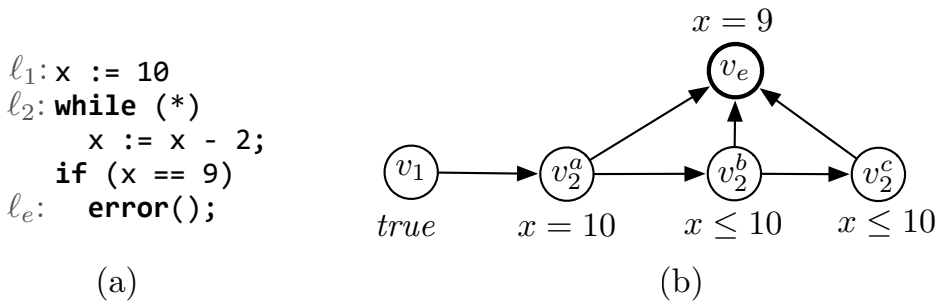


Figure 5.3: Example illustrating refinement with restricted DAG interpolants.

Given an ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$ with an error node v_{err}, it first constructs an edge labeling $\mathcal{L}_E$ using a BMC encoding such that for each ARG edge e, $\mathcal{L}_E(e)$ is the semantics of the corresponding action $\tau(e)$ (i.e., $[\![\tau(e)]\!]$), with variables renamed and added as necessary, and such that for any path $v_1, \ldots, v_k$, the formula $\bigwedge_{i\in[1,k)} \mathcal{L}_E(v_i, v_{i+1})$ encodes all executions from v_1 to v_k. Many BMC encodings can be used for this step—we use the approach of [GCS11]. For example, for the three edges (v_1, v_2^a), (v_2^a, v_e), (v_2^a, v_2^b) of the ARG in Figure 5.2(c), the $\mathcal{L}_E$ map is

$$\mathcal{L}_E(v_1, v_2^a) \equiv x_0 = 0 \wedge y_0 = 0, \tag{5.5}$$

$$\mathcal{L}_E(v_2^a, v_e) \equiv x_\phi \geqslant 4 \wedge y_\phi \leqslant 2 \wedge x_\phi = x_0 \wedge y_\phi = y_0, \text{ and} \tag{5.6}$$

$$\begin{aligned}\mathcal{L}_E(v_2^a, v_2^b) \equiv\ & (x_1 = x_0 + 1 \wedge y_1 = y_0 + 1) \vee \\ & (x_0 \geqslant 4 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1) \vee \\ & (x_1 = x_0 \wedge y_1 = y_0),\end{aligned} \tag{5.7}$$

where, in addition to renaming, two extra variables x_ϕ and y_ϕ were added for the SSA encoding since node v_e has multiple edges incident on it. $\mathcal{L}_E(v_1, v_2^a) \wedge \mathcal{L}_E(v_2^a, v_e)$ encodes all executions on the path v_1, v_2^a, v_e, and $\mathcal{L}_E(v_1, v_2^a) \wedge \mathcal{L}_E(v_2^a, v_2^b)$ encodes all executions on the path v_1, v_2^a, v_2^b. Second, the refined labels are computed as a DAG interpolant $\text{DItp} = \textsc{ComputeDItp}((V, E, v_{\mathsf{en}}, v_{\mathsf{err}}), \mathcal{L}_E)$. Note that after reversing the renaming done by BMC encoding (i.e., removing the subscripts), the DI DItp is a safe (by condition 2 of Definition 5.3) well-labeling (by condition 1 of Definition 5.3) of the ARG $\mathcal{A}$. Furthermore, $\text{DItp}(v)$ is expressed completely in terms of variables defined before and used after $v \in V$. The result of refinement on our running example is shown in Figure 5.2(d).

Using Ufo Refinement with Vinta While Vinta can use Ufo's refinement since it satisfies the specification of Refine in Definition 5.1, we found that it does not scale in practice. We believe there are two key reasons for this.

The first reason is that the DI-based refinement uses just the ARG while completely ignoring its node labeling (i.e., the set of reachable states discovered by AI). Thus, while the DI-based refinement recovers from imprecision to remove false alarms, it may introduce imprecision for further exploration steps. For example, consider the program in Figure 5.3(a) and its ARG in Figure 5.3(b) produced by AI using the Box domain. The ARG has a false alarm (in reality, v_e is unreachable). A possible DI-based refinement changes the labels of v_2^b, v_2^c, and v_e to $x \leqslant 10 \wedge x \neq 9$, $x \neq 9$, and *false*, respectively. While this is sufficient to eliminate the false alarm, the new labels do not form an inductive invariant, and

therefore further unrolling of the ARG is required. Note that the refinement "improved" the label of v_2^c to $x \neq 9$, but "lost" an important fact $x \leqslant 10$. Instead, we propose to restrict refinement to produce new labels that are stronger than the existing ones. In this example, such a restricted refinement would change the labels of v_2^b, v_2^c, and v_e to $x \leqslant 10 \wedge x \neq 9$, $x \leqslant 10 \wedge x \neq 9$, and *false*, thus resulting in a safe inductive invariant.

The second reason is that ARGs produced by AI are large, and generating interpolants directly from them takes too long. Here, again, part of the problem is that refinement does not use the existing labeling to simplify the constraints. Instead of computing a DI of the ARG, we propose to compute an RDI restricted by the current labeling. Since an RDI is simpler (i.e., weaker, has fewer connectives, etc.) than a corresponding DI, the hope is that it is also easier to compute.

Vinta Refinement Vinta's refinement procedure VintaRef is shown in Algorithm 6. It takes a labeled ARG $\mathcal{A}$ and returns a new safe well-labeling *labels* of $\mathcal{A}$. First, it encodes the edges of $\mathcal{A}$ using BMC encoding as described above (line 3). Second, the current labeling ψ of $\mathcal{A}$ is encoded to match the renaming introduced by the BMC encoding. For example, for v_2^a in our running example, $\psi(v_2^a) \equiv x = 0 \wedge y = 0$, and the encoding $\mathcal{L}_V(v_2^a) \equiv x_0 = 0 \wedge y_0 = 0$. Third, it uses ComputeRDItp (shown in Algorithm 6) to compute an RDI of $\mathcal{A}$ restricted by $\mathcal{L}_V$. Fourth, it turns the RDI into a DI by conjoining it with $\mathcal{L}_V$ (line 7). Finally, it decodes the labels by undoing the BMC encoding (line 9).

The function ComputeRDItp computes an RDI by reducing it to computing DAG interpolants, which can be computed using the procedure from Chapter 3. Note that it requires that $\mathcal{L}_V$ is a well-labeling, i.e., for all $(u, v) \in E$, $\mathcal{L}_V(u) \wedge \mathcal{L}_E(u, v) \Rightarrow \mathcal{L}_V(v)$. The idea is to "communicate" to the SMT solver the restriction of node u by conjoining $\mathcal{L}_V(u)$ to every edge from u. This information might be helpful to the SMT solver for simplifying its proofs[2] and the resulting interpolants.

Theorem 5.2 (Correctness of VintaRef)**.** VintaRef *satisfies the specification of* Refine *in Definition 5.1.*

There is a simple generalization of VintaRef: ψ on line 3 can be replaced by any over-approximation U of reachable states. The current invariant represented by the ARG is a good candidate and so are invariants computed by other techniques. The only restriction is that ComputeRDItp requires U to be a well-labeling. Removing this restriction from ComputeRDItp remains an open problem.

5.5 Experimental Evaluation

We have implemented Vinta in the UFO_{app} framework (Chapter 7) for verifying C programs, which is built on top of the LLVM compiler infrastructure [LA04]. Our implementation is an extension of our implementation of the Ufo algorithm. Vinta's implementation allows abstract domains to be easily plugged in and experimented with. In the rest of this section, we describe our experimental setup and evaluation.

Abstraction Functions We are using a simple abstraction function to convert between Boolean expressions and Boxes and Box abstract domains. Given a formula φ, we first convert it to *Negation Normal Form* (NNF), where negations only appear at the level of literals. Then, we replace all literals

[2]The abstract interpretation results conjoined to the formulas may help the SMT solver discover useful *theory lemmas* and prove unsatisfiability more efficiently.

Algorithm 6 Vinta's refinement technique.

```
1: function VintaRef(ARG A = (V, E, v_en, ν, τ, ψ))
2:     L_E ← EncodeBmc(A)
3:     L_V ← Encode(ψ)
4:     RDItp ← ComputeRDItp((V, E, v_en, v_err), L_E, L_V)
5:     if RDItp = ∅ then
6:         return RDItp
7:     for all v ∈ V do
8:         RDItp(v) ← RDItp(v) ∧ L_V(v)
9:     return DecodeBmc(RDItp)

Require: L_V is a well-labeling of G
10: function ComputeRDItp(G, L_E, L_V)
11:     for all e = (u, v) ∈ E do
12:         L_E(e) ← L_V(u) ∧ L_E(e)
13:     RDItp ← ComputeDItp(G, L_E)
14:     return RDItp
```

Algorithm	#Solved	#Safe	#Unsafe	Total Time (s)
vBox	**71**	20	**51**	**580** (539/**41**)
uBox	68	19	49	1,240 (1,162/78)
vBoxes	67	25	42	1,782 (596/1,186)
uBoxes	60	18	42	2,731 (808/1,923)
CpaAbe	65	**29**	36	1,167 (707/460)
CpaMemo	64	24	40	1,794 (**454**/1,341)
iUfo	70	20	50	1,535 (1,457/78)
cpUfo	69	19	50	1,687 (1,509/178)
bpUfo	64	15	49	1,062 (57/1,006)

Table 5.1: Evaluation of Vinta: results summary.

involving more than one variable (e.g., $x + y = 0$) with *true*, thus over-approximating φ and removing all terms not expressible in Box. Finally, for Box, we additionally use join to approximate logical disjunction. This naive approach is very imprecise in general, but works well on our benchmarks.

Experimental Setup For evaluation, we used `ntdrivers-simplified`, `ssh-simplified`, and `systemc` benchmarks from the 2012 Software Verification Competition (SV-COMP 2012) [Bey12]. In total, we had 93 C programs (41 safe and 52 buggy).

We implemented several instantiations of Vinta:

- vBox: Vinta using Box,
- vBoxes: Vinta using Boxes,
- uBox: Vinta using Box and UfoRef for refinement,
- uBoxes, Vinta using Boxes and UfoRef.

vBox and vBoxes used VintaRef for refinement. For Box and Boxes, we used the widening operators $\text{WidenWith}_{\sqcup}$ and $\text{WidenWith}_{\vee}$ from Section 5.4.2, respectively. In all cases, we applied widening on every third unrolling of each loop. We compared Vinta against the top two tools from SV-COMP

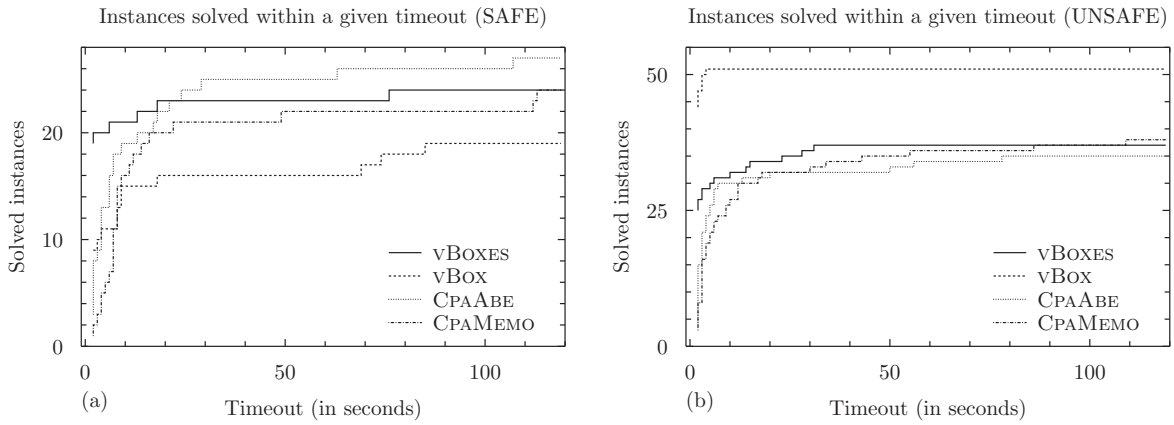


Figure 5.4: Evaluation of Vinta: Instances solved vs. timeout.

2012: CPAchecker-Abe [LW12] (CpaAbe) and CPAchecker-Memo [Won12] (CpaMemo), which are two variations of the predicate-abstraction-based software model checker CPAchecker [BK11]. For both tools, we used the same version and configuration as in the competition. We also compared against several instantiations of the Ufo algorithm: iUfo, cpUfo, and bpUfo, using interpolation-based verification by itself and in combination with Cartesian and Boolean predicate abstractions, respectively. (See Chapter 4 for a detailed description of these Ufo instantiations.)

The overall results are summarized in Table 5.1. All experiments were conducted on a 3.40GHz Intel Core i7 processor with 8GB of RAM running Ubuntu Linux v11.10. We imposed a time limit of 500 seconds and a memory limit of 4GB per program. For each tool, we show the number of safe and unsafe instances solved and the total time taken. For example, vBox solved 20 safe and 51 unsafe examples in 580 seconds, spending 539s on safe ones and 41s on unsafe ones (time spent in unsolved instances is not counted). vBox is an overall winner, and is able to solve the most unsafe instances in the least amount of time. CpaAbe is the winner on the safe instances, with vBoxes coming in second. In the rest of this section, we examine these results in more detail.

Instances Solved vs. Timeout Figure 5.4 shows the number of instances solved in a given timeout for (a) safe and (b) unsafe benchmarks, respectively. To avoid clutter, we omit iUfo, bpUfo, and cpUfo from the graphs and restrict the timeout to 120s, since only a few instances took more time. For the safe cases, vBoxes is a clear winner for the timeout of $\leqslant$ 10s. Indeed, on most safe benchmarks, vBoxes takes a lot less time to complete than CpaAbe, CpaMemo, and all other instantiations of Ufo and Vinta. For the unsafe cases, vBox is a clear winner for all timeouts. Interestingly, the extra precision of Boxes makes vBoxes perform poorly on unsafe instances: it either solves an unsafe instance in one iteration (i.e., no refinement), or runs out of time in the first AI- or refinement-phase.

Detailed Comparison We now examine a portion of the benchmark suite in more detail, specifically, safe `ssh-simplified` benchmarks and safe `token_ring` benchmarks (from `systemc`). Table 5.2 shows the time taken by the different instantiations of Vinta, CpaAbe, and CpaMemo. On these benchmarks, we observe that vBoxes outperforms all other approaches.

Compared with CpaAbe and CpaMemo, vBoxes is able to solve almost all instances in much less time. For example, on `token_ring.05`, both CpaAbe and CpaMemo fail to return a result, but vBoxes

Program	vBoxes	uBoxes	vBox	uBox	CpaAbe	CpaMemo
s3_clnt_1	**0.30**	**0.30**	8.61	13.67	7.34	11.63
s3_clnt_2	**0.3**	**0.30**	8.79	13.45	6.72	8.53
s3_clnt_3	0.30	**0.29**	9.01	6.80	9.72	7.10
s3_clnt_4	**0.30**	**0.30**	9.55	8.52	6.33	12.43
s3_srvr_1a	**0.15**	–	1.08	–	2.86	4.344
s3_srvr_1b	**0.02**	**0.02**	–	–	1.49	1.64
s3_srvr_1	**0.00**	**0.00**	**0.00**	**0.00**	21.21	8.63
s3_srvr_2	**0.64**	115.48	–	115.13	63.44	113.07
s3_srvr_3	**0.75**	123.57	69.70	123.61	17.23	22.55
s3_srvr_4	**0.59**	168.44	85.81	168.08	7.50	14.57
s3_srvr_6	473.15	319.00	**74.87**	359.39	181.82	–
s3_srvr_7	**13.82**	–	–	274.12	24.84	112.53
s3_srvr_8	**0.69**	78.53	245.52	76.12	18.48	8.82
token_ring.01	**0.94**	–	4.05	–	4.13	8.04
token_ring.02	**2.53**	–	18.29	–	6.69	49.11
token_ring.03	**6.06**	–	–	–	29.55	–
token_ring.04	**18.22**	–	–	–	146.43	–
token_ring.05	**76.29**	–	–	–	–	–
token_ring.06	–	–	–	–	–	–
token_ring.07	–	–	–	–	–	–
token_ring.08	–	–	–	–	–	–

Table 5.2: Evaluation of Vinta: detailed results and tool comparison.

proves safety in 76 seconds. Similarly, vBoxes is superior on most `ssh-simplified` examples.

To understand the importance of the refinement strategy, consider the `ssh-simplified` benchmarks. The invariant for most `ssh-simplified` instances is computable using Boxes with an appropriate widening strategy ("widen on every fourth unrolling"). The results in the table show how Vinta's refinement strategy is able to recover precision when an inadequate refinement strategy is used (i.e., "widen on every third unrolling"). Using Ufo's refinement, uBoxes takes substantially more time and more iterations or fails to return a result within the allotted time limit. For example, on `s3_srvr_2`, vBoxes requires a single refinement, whereas uBoxes requires 38. Positive effects of Vinta's AI-guided refinement are also visible in vBox vs. uBox.

In summary, our results demonstrate the power of Vinta's refinement strategy and show how basic instantiations of Vinta can compete and outperform highly-optimized verification tools like CPAchecker. To further improve Vinta's performance, it would be interesting to experiment with other abstract domains as well as with different automatic strategies for choosing an appropriate domain. For example, we saw that Boxes, in comparison with Box, generates very large ARGs for unsafe examples. One strategy would be to keep track of ARG size and time spent in refinement and revert to a less precise abstract domain like Box when they become too large.

5.6 Related Work

Vinta is closely related to the Dagger tool of Gulavani et al. [GCNR08] that is also based on refining AI, and to our earlier tool Ufo that combines predicate abstraction with interpolation-based verification. The key differences between Vinta and Dagger are: (1) Dagger can only refine imprecision caused

by widening and join. Vinta can refine imprecision up to the concrete semantics of the program (as modeled in SMT). (2) Dagger refines joins explicitly, which may result in an exponential increase in the number of abstract states compared to the size of the program. Vinta refines joins implicitly using interpolants and SMT. (3) Dagger requires a specialized interpolation procedure, which, so far, has only been developed for the octagon and the polyhedra domains. Vinta can use any off-the-shelf interpolating SMT solver, immediately benefiting from any advances in the field.

Compared to Ufo, Vinta improves both the exploration algorithm (by extending it to an arbitrary abstract domain) and the refinement procedure (by extending it to use intermediate invariants computed by AI). Both of these extensions are important for Vinta's success, as shown in the experiments in Section 5.5.

5.7 Conclusion

Invariant generation via abstract interpretation is one of the most scalable techniques for computing inductive invariants. The price of efficiency is false alarms created by weak invariants that fail to preclude unsafe program states. In this chapter, we described Vinta, a refinement algorithm for general invariant generation with abstract domains that recovers lost precision due to the different over-approximations employed by abstract interpretation: inexpressive abstract domains, approximate joins, and widening. Vinta is an improvement and generalization of Ufo (Chapter 4). Specifically, Vinta allows Ufo to adopt general abstract domains (as opposed to predicate abstraction) by incorporating a novel widening strategy. Additionally, Vinta adopts an extension of DAG interpolants for refinement, which allows it to utilize results of abstract interpretation to compute "better" interpolants and improve SMT solving. Our extensive evaluation demonstrates (1) the improvements Vinta makes over Ufo and state-of-the-art tools and (2) the utility of its new refinement technique.

Chapter 6

Interpolation-based Interprocedural Verification

"My language is a metaphor for the metaphor."

— Mahmoud Darwish

6.1 Introduction

In the previous chapters, we discussed new interpolation-based verification techniques, all of which have been intraprocedural—constrained to programs with no procedure calls and recursion. Despite the promise of interpolation-based techniques and the mass of literature on the subject, the question of adapting interpolation-based verification to the interprocedural setting has received little attention [McM10, HHP10].

In this chapter, we present Whale: an interprocedural IB algorithm that produces modular safety proofs of (recursive) sequential programs.[1] Our key insight is to use interpolation to compute a function summary by generalizing from an under-approximation of a function, thus avoiding the need to fully expand the function and resulting in modular proofs of correctness. The use of interpolants allows us to produce concise summaries that eliminate facts irrelevant to the property in question. We also show how the power of SMT solvers can be exploited in our setting by encoding a path condition over multiple (or all) interprocedural paths of a program in a single formula. We have implemented a prototype of Whale using the LLVM compiler infrastructure [LA04] and verified properties of low-level C code written for the pacemaker grand challenge.

Whale can be viewed as an adaptation of lazy abstraction with interpolants (LAWI) [McM06] to the interprocedural setting. LAWI computes a safe inductive invariant by unrolling the control-flow graph of a given procedure and annotating it with a Hoare-style proof of correctness, with the hope that the proof of a bounded unrolling is a proof for the whole program. Whale lifts this idea to the granularity of procedures. Specifically, Whale unrolls the call graph of a program with procedures (and possibly recursion). Given a bounded unrolling of a call graph, we can encode all paths through the unrolling as

[1] The naming is a subtle pun on McMillan's Impact (aka LAWI) tool and algorithm [McM06]: a Whale makes a big Impact.

an SMT formula, and use an SMT solver to check if there are bugs. When no counterexamples are found, we demonstrate how *state/transition interpolants*—a new notion of interpolants that we present—can be used to annotate the call graph unrolling with procedure summaries. In a manner analogous to LAWI, the goal is to compute annotations for the bounded unrolling that are procedure summaries for the complete (unbounded) procedures.

Contributions

We summarize this chapter's contributions as follows:

- We introduce a reformulation of McMillan's lazy abstraction with interpolants algorithm [McM06] to the interprocedural setting.
- We introduce the notion of state/transition interpolants for guessing procedure summaries, and show how they can be computed using existing interpolant generation techniques.
- We show that our prototype implementation can outperform existing tools for verification of low-level code from the artificial cardiac pacemaker challenge.

Organization

This chapter is organized as follows:

- In Section 6.2, we demonstrate the operation of Whale on a simple recursive program.
- In Section 6.3, we formalize concepts required for the rest of the chapter.
- In Section 6.4, we present an extension of abstract reachability graphs for proving correctness of programs with procedures and recursion.
- In Section 6.5, we formalize Whale and discuss its properties.
- In Section 6.6, we present an experimental evaluation of Whale.
- In Section 6.7, we place Whale within related work on interprocedural analysis and interpolation-based techniques.
- Finally, in Section 6.8, we conclude the chapter with a summary and a comparison of Whale with Ufo and Vinta.

6.2 Illustrative Example

In this section, we use Whale to prove that `mc91` in Figure 6.1, a variant of the famous McCarthy 91 function [MM70], always returns a value $\geqslant 91$, i.e., `mc91(p)` $\geqslant 91$ for all values of p.

Whale works by iteratively constructing a forest of abstract reachability graphs (ARGs) (we call it an *iARG*) with one ARG for the main function and one ARG for each function call inside each ARG. Each ARG $\mathcal{A}_i$ is associated with some function F_k; an expression G_i over the arguments of F_k, called the *guard*; and an expression S_i over the arguments and the return variables of F_k, called the *summary*. ***Note that our definition of ARGs in this chapter is slightly different from that***

in Chapter 2 in order to accommodate for parameters and returns of functions. Intuitively, Whale uses ARG $\mathcal{A}_i$ to show that function F_k behaves according to S_i, assuming the arguments satisfy G_i and assuming all other functions behave according to their corresponding ARGs in the iARG. A node v in an ARG $\mathcal{A}_i$ corresponds to a control location ℓ_v and is labeled by a Boolean formula e_v over program variables. Whale maintains the invariant that e_v is an over-approximation of the states reachable from those in G_i, at the entry point of F_k, along the path to v. It is always sound to let e_v be *true*. We now apply Whale to `mc91` in Figure 6.1, producing ARGs `A` (starting with $\mathtt{A}_1$), with `G` and `S` as their guards and summaries, respectively.

Step 1 For each ARG in Figure 6.1, the number inside a node v is the location ℓ_v and the expression in braces is e_v. For our property, $\mathtt{mc91(p)} \geqslant 91$, the guard $\mathtt{G}_1$ is *true* and the summary $\mathtt{S}_1$ is $r \geqslant 91$. The single path of $\mathtt{A}_1$ is a potential counterexample: it reaches the return statement (line 8), and node 8 is labeled *true* (which does not imply the summary $r \geqslant 91$). To check for feasibility of the computed counterexample, Whale checks satisfiability of the corresponding *path formula*

$$\pi = true \wedge (p > 100) \wedge (r = p - 10) \wedge (r < 91)$$

obtained by conjoining the guard, all of the conditions and assignments on the path, and the negation of the summary. Here, π is unsatisfiable. Hence, the counterexample is infeasible, and the ARG labeling can be strengthened to exclude it.

Step 2 Like [McM06], Whale uses interpolants to strengthen the labels. Recall that for a pair of formulas (A, B) such that $A \wedge B$ is unsatisfiable, an interpolant I is a formula in the common vocabulary of A and B such that $A \Rightarrow I$ and $I \Rightarrow \neg B$. Intuitively, I is a weakening of A that is inconsistent with B. Each node v in the infeasible counterexample is labeled by an interpolant obtained by letting A be the part of the path formula for the path from root to v, and B be the rest of the path formula. The new labeling is shown in Figure 6.1 in ARG $\mathtt{A}'_1$.

Step 3 Next, the second path through `mc91` is added to $\mathtt{A}'_1$ and has to be checked for feasibility. This path has two recursive calls that need to be represented in the path formula. For each call statement, Whale creates a new *justifying* ARG, in order to keep track of the under-approximation of the callee used in the proof of the caller and to construct the proof that the callee behaves according to a given specification.

Let $\mathtt{A}_2$ and $\mathtt{A}_3$ be the ARGs justifying the first and the second calls, respectively. For simplicity of presentation, assume that $\mathtt{A}_2$ and $\mathtt{A}_3$ have been unrolled and are identical to $\mathtt{A}_1$ in Figure 6.1. The path formula π for the path 2, 5, ..., 8a is constructed by under-approximating the callees by inlining them with the justifying ARGs (shown by bold labels on the grey call-edges in $\mathtt{A}'_1$). Specifically,

$$\pi = true \wedge (p \leqslant 100) \wedge (p_1 = p + 11) \wedge U_1 \wedge U_2 \wedge (r < 91),$$

where U_1 and U_2 represent the under-approximations of the called functions on edges (6,7) and (7,8), respectively. This path formula is unsatisfiable and thus the counterexample is infeasible. Again, interpolants are used to strengthen node labels, as shown in ARG $\mathtt{A}''_1$. Furthermore, the interpolants are also used to generalize the under-approximations of the callees by taking the interpolant of the pair

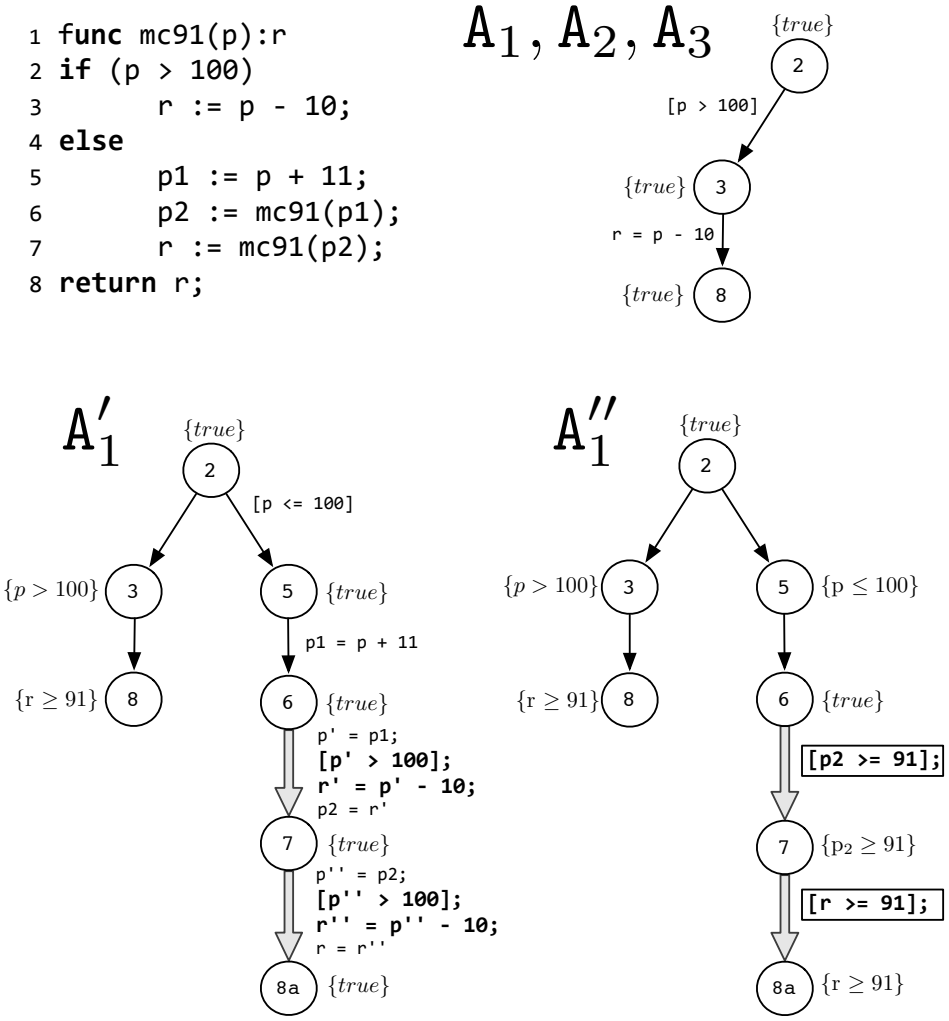


Figure 6.1: Illustration of Whale on the McCarthy 91 function.

(A, B), where A is the path formula of the under-approximation and B is the rest of the path formula. The resulting interpolant I is a specification of the callee that is weaker than its under-approximation, but strong enough to exclude the infeasible counterexample. For example, to generalize the under-approximation U_1, we set A to U_1 and B to $true \wedge (p \leqslant 100) \wedge (p_1 = p + 11) \wedge U_2 \wedge (r < 91)$. The resulting generalizations, which happen to be $r \geqslant 91$ for both calls, are shown on the call-edges in ARG $\mathtt{A}_1''$ with variables renamed to suit the call context.

Step 4 At this point, all intraprocedural paths of `mc91` have been examined. Hence, $\mathtt{A}_1''$ is a proof that the body of `mc91` returns $r \geqslant 91$ assuming that the first call returns $r \geqslant 91$ and that the second one returns $r \geqslant 91$ whenever $p \geqslant 91$. To discharge the assumptions, Whale sets guards and summaries for the ARGs $\mathtt{A}_2$ and $\mathtt{A}_3$ as follows: $\mathtt{G}_2 = true$, $\mathtt{S}_2 = r \geqslant 91$, $\mathtt{G}_3 = p \geqslant 91$ and $\mathtt{S}_3 = r \geqslant 91$, and can continue to unroll them following steps 1-3 above. However, in this example, the assumptions on recursive calls to `mc91` are weaker than what was established about the body of `mc91`. Thus, we conclude that the ARGs $\mathtt{A}_2$ and $\mathtt{A}_3$ are *covered* by $\mathtt{A}_1''$ and do not need to be expanded further, finishing the analysis. Intuitively,

the termination condition is based on the Hoare proof rule for recursive functions [Hoa71].

In practice, Whale only keeps track of guards, summaries, and labels at entry and exit nodes. Other labels can be derived from those when needed.

Summary Whale explores the program by unwinding its control flow graph. Each time a possible counterexample is found, it is checked for feasibility and, if needed, the labels are strengthened using interpolants. If the counterexample is interprocedural, then an under-approximation of the callee is used for the feasibility check, and interpolants are used to guess a summary of the called function. Whale attempts to verify the summary in a similar manner, but if the verification is unsuccessful, it generates a counterexample which is used to refine the under-approximation used by the caller and to guess a new summary.

6.3 Preliminaries: Procedural Programs and Hoare Proofs

This section presents important definitions required in the rest of the chapter. Specifically, we extend and modify the definition of programs from Chapter 2 to contain procedures and procedure calls. We also review some Hoare logic rules for reasoning about procedure calls and recursion.

Program Syntax We divide program statements into simple statements and function calls. A *simple statement* is either an assignment statement `x := E` or a conditional statement `assume(Q)`, where `x` is a program variable, and E and Q are an expression and a Boolean expression over program variables, respectively. We write $[\![T]\!]$ for the standard semantics of a simple statement T.

Functions are declared as

$$\texttt{func foo}\ (p_1, \ldots, p_n) : r_1, \ldots, r_k \quad B_{\texttt{foo}},$$

defining a function with name `foo`, n parameters $\mathcal{P} = \{p_1, \ldots, p_n\}$, k return variables $\mathcal{R} = \{r_1, \ldots, r_k\}$, and body $B_{\texttt{foo}}$. We assume that a function never modifies its parameters. The return value of a function is the valuation of all return variables at the time when the execution reaches the exit location. Functions are called using syntax

$$b_1, \ldots, b_k = \texttt{foo}\ (a_1, \ldots, a_n)$$

interpreted as a call to `foo`, passing *values* of local variables $a_1, \ldots, a_n$ as parameters $p_1, \ldots, p_n$, respectively, and storing the *values* of the return variables $r_1, \ldots, r_k$ in local variables $b_1, \ldots, b_k$, respectively. The variables $\{a_i\}_{i=1}^n$ and $\{b_i\}_{i=1}^k$ are assumed to be disjoint. Moreover, for all $i, j \in [1, n]$, such that $i \neq j$, we have $a_i \neq a_j$. That is, there are no duplicate elements in $\{a_i\}_{i=1}^n$. The same holds for the set $\{b_i\}_{i=1}^k$.

Program Model A *program* $P = (F_1, F_2, \ldots, F_n)$ is a list of n functions. Each *function* F is a tuple $(\mathcal{L}, \delta, \mathsf{en}, \mathsf{ex}, \mathcal{P}, \mathcal{R}, \mathsf{Var})$, where

- $\mathcal{L}$ is a finite set of control locations,
- δ is a finite set of actions,
- $\mathsf{en}, \mathsf{ex} \in \mathcal{L}$ are designated entry and exit locations, respectively, and

$$\frac{P' \Rightarrow P \quad \{P\}T\{Q\} \quad Q \Rightarrow Q'}{\{P'\}T\{Q'\}}$$

$$\frac{(P' \wedge \vec{p} = \vec{a}) \Rightarrow P \quad \{P\}B_F\{Q\} \quad (Q \wedge \vec{p}, \vec{r} = \vec{a}, \vec{b}) \Rightarrow Q'}{\{P'\}\vec{b} = F(\vec{a})\{Q'\}}$$

$$\frac{\{P\}\vec{b} = F(\vec{a})\{Q\} \quad \vdash \quad \{P\}B_F\{Q\}}{\{P\}\vec{b} = F(\vec{a})\{Q\}}$$

Figure 6.2: Hoare logic rules for interprocedural reasoning.

- $\mathcal{P}$, $\mathcal{R}$ and Var are sets of parameter, return and local variables, respectively (we use no global variables).

An *action* $(\ell_1, T, \ell_2) \in \delta$ is a tuple where $\ell_1, \ell_2 \in \mathcal{L}$ and T is a program statement over $\mathsf{Var} \cup \mathcal{P} \cup \mathcal{R}$. We assume that the control-flow graph (CFG) represented by $(\mathcal{L}, \delta)$ is a directed acyclic graph (DAG), where loops are modeled by tail recursion. Execution starts in the first function in the program. For a function $F = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{ex}, \mathcal{P}, \mathcal{R}, \mathsf{Var})$, we write $\mathcal{L}(F)$ for $\mathcal{L}$, $\delta(F)$ for δ, etc. We write $\vec{p}_i$ and $\vec{r}_i$ to denote vectors of parameter and return variables of F_i.

Floyd-Hoare Logic A *Hoare Triple* [Hoa69] $\{P\}T\{Q\}$, where T is a program statement and P and Q are propositional formulas, indicates that if P is true of program variables before executing T, and T terminates, then Q is true after T completes. P and Q are called the *pre-* and the *postcondition*, respectively.

We make use of three proof rules shown in Figure 6.2:

- The first is the *rule of consequence*, indicating that a precondition of a statement can be strengthened whereas its postcondition can be weakened.
- The second is the *rule of function instantiation*, where B_F is a body of a function F with parameters $\vec{p}$ and returns $\vec{r}$. It explicates the conditions under which F can be called with actual parameters $\vec{a}$, returning $\vec{b}$, and with P' and Q' as pre- and postconditions, respectively. For this rule, we assume that P is over the set of variables $\vec{p}$ and Q is over the variables $\vec{p}$ and $\vec{r}$.
- The third is the *rule of recursion*, indicating that a recursive function F satisfies the pre-/postconditions (P, Q) if the body of F satisfies (P, Q) *assuming* that all recursive calls satisfy (P, Q).

For two sets of triples X and Y, $X \vdash Y$ indicates that Y can be proven from X (i.e., X is weaker than Y). We also say $\vdash X$ to mean that X is valid, i.e., that it follows from the axioms.

6.4 Interprocedural Reachability Graphs

In this section, we extend and modify the definition of abstract reachability graphs from Chapter 2 to handle procedure calls and define safety over pre-/post-conditions as opposed to error locations. At a high level, an ARG represents an exploration of the state space of a function, while making assumptions about the behavior of other functions it calls. We define a forest of ARGs, called an *Interprocedural Abstract Reachability Graph* (iARG), to represent a modular proof of correctness of a program with multiple functions.

Abstract Reachability Graphs (ARGs) Let $F = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{ex}, \mathcal{P}, \mathcal{R}, \mathsf{Var})$ be a function. A *Reachability Graph* (RG) of F is a tuple $(V, E, \epsilon, \nu, \tau)$ where

- (V, E, ϵ) is a DAG rooted at $\epsilon \in V$,
- $\nu : V \to \mathcal{L}$ is a *node map*, mapping nodes to control locations such that $\nu(\epsilon) = \mathsf{en}$ and $\nu(v) = \mathsf{ex}$ for every leaf node v,
- and $\tau : E \to \delta$ is an *edge map*, mapping edges to program actions such that for every edge $(u, v) \in E$ there exists $(\nu(u), \tau(u, v), \nu(v)) \in \delta$.

We write $V^e = \{v \in V \mid \nu(v) = \mathsf{ex}\}$ for all leaves (exit nodes) in V. We call an edge e, where $\tau(e)$ is a call statement, a *call-edge*. We assume that call-edges are ordered in some linearization of a topological order of (V, E).

An *Abstract Reachability Graph* (ARG) $\mathcal{A}$ of F is a tuple (U, ψ, G, S), where

- U is reachability graph of F,
- ψ is a *node labeling* that labels the root and leaves of U with formulas over program variables,
- G is a formula over $\mathcal{P}$ called a *guard*,
- and S is a formula over $\mathcal{P} \cup \mathcal{R}$ called a *summary*.

For example, ARG $\mathtt{A}_1$ is given in Figure 6.1 with a guard $\mathtt{G}_1 = \mathit{true}$, a summary $\mathtt{S}_1 = r \leqslant 91$, and with ψ shown in braces.

An ARG $\mathcal{A}$ is *complete* if and only if for every path in F there is a corresponding path in $\mathcal{A}$. Specifically, $\mathcal{A}$ is complete if and only if every node $v \in V$ has a successor for every action $(\nu(v), T, \ell) \in \delta$, i.e., there exists an edge $(v, w) \in E$ such that $\nu(w) = \ell$ and $\tau(v, w) = T$. It is *safe* if and only if for every leaf $v \in V$, $\psi(v) \Rightarrow S$. For example, in Figure 6.2, ARG $\mathtt{A}_1''$ is safe and complete, ARG $\mathtt{A}_1'$ is complete but not safe, and other ARGs are neither safe nor complete.

Interprocedural ARGs An *Interprocedural Abstract Reachability Graph* (iARG) $\mathcal{I}^{\mathcal{A}}(P)$ of a program $P = (F_1, \ldots, F_n)$ is a tuple $(\sigma, \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}, R^{\mathcal{J}}, R^{\mathcal{C}})$, where

- $\sigma : [1, k] \to [1, n]$ maps ARGs to corresponding functions, i.e., $\mathcal{A}_i$ is an ARG of $F_{\sigma(i)}$,
- $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ is a set of ARGs,
- $R^{\mathcal{J}}$ is an acyclic *justification relation* between ARGs such that $(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}, R^{\mathcal{J}})$ is the *justification tree* of $\mathcal{I}^{\mathcal{A}}(P)$ rooted at $\mathcal{A}_1$,
- and $R^{\mathcal{C}}$ is a *covering relation* between ARGs.

The justification tree corresponds to a partially unrolled call graph. Informally, if $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{J}}$ then there is a call-edge in $\mathcal{A}_i$ that is *justified* (expanded) by $\mathcal{A}_j$. We write $\mathcal{A}_i \sqsubseteq_{\mathcal{J}} \mathcal{A}_j$ for the ancestor relation in the justification tree. Given two nodes $u, v \in V_i$, an interprocedural (u, v)-path in $\mathcal{A}_i$ is a (u, v)-path in $\mathcal{A}_i$ in which every call-edge e is expanded, recursively, by a trace in an ARG $\mathcal{A}_j$, where $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{J}}$. For convenience, we assume that $\sigma(1) = 1$, and use a subscript to refer to components of an $\mathcal{A}_i$ in $\mathcal{I}^{\mathcal{A}}(P)$, e.g., ψ_i is the node labeling of $\mathcal{A}_i$.

An ARG $\mathcal{A}_i$ is *directly covered* by $\mathcal{A}_j$ if and only if $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{C}}$. $\mathcal{A}_i$ is *covered* by $\mathcal{A}_j$ if and only if $\mathcal{A}_j \sqsubseteq_{\mathcal{J}} \mathcal{A}_i$ and $\mathcal{A}_j$ is directly covered by another ARG. $\mathcal{A}_i$ is covered if and only if it is covered by some $\mathcal{A}_j$; otherwise, it is *uncovered*. A covering relation $R^{\mathcal{C}}$ is *sound* if and only if for all $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{C}}$:

- $\mathcal{A}_i$ and $\mathcal{A}_j$ are mapped to the same function F_l, i.e., $\sigma(i) = \sigma(j) = l$;
- $i \neq j$ and $\mathcal{A}_i$ is not an ancestor of $\mathcal{A}_j$, i.e., $\mathcal{A}_i \not\sqsubseteq_{\mathcal{J}} \mathcal{A}_j$;
- the specification of $\mathcal{A}_j$ is stronger than that of $\mathcal{A}_i$, i.e., $\{G_j\}\vec{r} = F_l(\vec{p})\{S_j\} \vdash \{G_i\}\vec{r} = F_l(\vec{p})\{S_i\}$;
- and $\mathcal{A}_j$ is uncovered.

For example, for ARGs in Figure 6.1, $(\mathtt{A_3}, \mathtt{A}_1'') \in R^{\mathcal{C}}$, and $\mathtt{A}_1''$ is uncovered. $\mathtt{A_3}$ is left incomplete, since the validity of its guard and summary follow from the validity of the guard and summary of $\mathtt{A}_1''$:

$$\{true\}B_{\mathtt{mc91}}\{r \geqslant 91\} \vdash \{p \geqslant 91\}B_{\mathtt{mc91}}\{r \geqslant 91\},$$

where $(true, r \geqslant 91)$ and $(p \geqslant 91, r \geqslant 91)$ are the guard and summary pairs of $\mathtt{A}_1''$ and $\mathtt{A_3}$, respectively.

An iARG $\mathcal{I}^{\mathcal{A}}(P)$ is *safe* if and only if $\mathcal{A}_1$ is safe. It is *complete* if and only if every uncovered ARG $\mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P)$ is complete.

6.5 The Whale Algorithm

In this section, we provide a detailed exposition of Whale. We begin with an overview of its basic building blocks.

Overview Given a program $P = (F_1, \ldots F_n)$ and a pair of formulas (G, S), our goal is to decide whether $\vdash \{G\}B_{F_1}\{S\}$. Whale starts with an iARG $\mathcal{I}^{\mathcal{A}}(P) = (\sigma, \{\mathcal{A}_1\}, R^{\mathcal{J}}, R^{\mathcal{C}})$ where $\sigma(1) = 1$, and $R^{\mathcal{J}}$ and $R^{\mathcal{C}}$ are empty relations. $\mathcal{A}_1$ has one vertex v and $\nu(v) = \mathsf{en}(F_1)$. The guard G_1 and summary S_1 are set to G and S, respectively. In addition to the iARG, Whale maintains a map $\mathcal{J}$ from call-edges to ARGs and an invariant that $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{J}}$ if and only if there exists $e \in E_i$ such that $\mathcal{J}(e) = \mathcal{A}_j$.

Whale is an extension of Impact [McM06] to interprocedural programs. Its three main operations (shown in Algorithm 7), ExpandARG, CoverARG, and RefineARG, correspond to their counterparts of Impact. ExpandARG adds new paths to explore, CoverARG ensures that there is no unnecessary exploration, and RefineARG checks for presence of counterexamples and guesses guards and summaries. All operations maintain soundness of $R^{\mathcal{C}}$. Whale terminates either when RefineARG finds a counterexample, or when none of the operations are applicable. In the latter case, the iARG is complete. We show at the end of this section that this also establishes the desired result: $\vdash \{G_1\}B_{F_1}\{S_1\}$.

ExpandARG adds new paths to an ARG $\mathcal{A}_i$ if it is incomplete, by replacing an RG U_i with a supergraph U_i'. Implicitly, new ARGs are created to justify any new call-edges, as needed, and are logged in the justification map $\mathcal{J}$. A new ARG $\mathcal{A}_j$ is initialized with a $G_j = S_j = true$ and $V_j = \{v\}$, where v is an entry node. The paths can be added one-at-a-time (as in Impact and in the example in Section 6.2), all-at-once (by adding a complete CFG), or in other ways. Finally, all affected labels are reset to *true*.

CoverARG covers an ARG $\mathcal{A}_i$ by $\mathcal{A}_j$. Its precondition maintains the soundness of $R^{\mathcal{C}}$. Furthermore, we impose a total order, $\prec$, on ARGs such that $\mathcal{A}_i \sqsubset \mathcal{A}_j$ implies $\mathcal{A}_i \prec \mathcal{A}_j$, to ensure that CoverARG

Algorithm 7 The WHALE Algorithm.

Require: $\mathcal{A}_i$ is uncovered and incomplete
1: **function** EXPANDARG(ARG $\mathcal{A}_i$)
2: replace U_i with a supergraph U_i',
where U_i is the unwinding of $\mathcal{A}_i$
3: RESET($\mathcal{A}_i$)

Require: $\mathcal{A}_i \not\sqsubseteq_{\mathcal{J}} \mathcal{A}_j$, $\sigma(i) = \sigma(j)$,
$\mathcal{A}_i$ and $\mathcal{A}_j$ are uncovered,
$\{G_j\}B_{F_{\sigma(i)}}\{S_j\} \vdash \{G_i\}B_{F_{\sigma(i)}}\{S_i\}$
4: **function** COVERARG(ARGs $\mathcal{A}_i$ and $\mathcal{A}_j$)
5: $R^{\mathcal{C}} \leftarrow R^{\mathcal{C}} \setminus \{(\mathcal{A}_l, \mathcal{A}_i) \mid (\mathcal{A}_l, \mathcal{A}_i) \in R^{\mathcal{C}}\}$
6: $R^{\mathcal{C}} \leftarrow R^{\mathcal{C}} \cup \{(\mathcal{A}_i, \mathcal{A}_j)\}$

7: **function** RESET(ARG $\mathcal{A}_i$)
8: $\forall v \cdot \psi_i(v) \leftarrow true$
9: **for all** $\{\mathcal{A}_j \mid \exists e \in E_i \cdot \mathcal{J}(e) = \mathcal{A}_j\}$ **do**
10: $(G_j, S_j) \leftarrow (true, true)$
11: RESET($\mathcal{A}_j$)

12: **function** UPDATE(ARG $\mathcal{A}_i$, g, s)
13: $(G_i, S_i) \leftarrow (G_i \wedge g, S_i \wedge s)$
14: RESET($\mathcal{A}_i$)

Require: $\mathcal{A}_i$ is uncovered,
$\nu(v) = \mathsf{ex}(F_{\sigma(i)})$,
$\psi_i(v) \not\Rightarrow S_i$
15: **function** REFINEARG(vertex v in $\mathcal{A}_i$)
16: $cond \leftarrow G_i \wedge$ IARGCOND($\mathcal{A}_i, \{v\}$) $\wedge \neg S_i$
17: **if** $cond$ is unsatisfiable **then**
18: $g_0, s_0, g_1, s_1, \ldots, s_{m+1} \leftarrow$STITP($cond$)
19: $\psi_i(v) \leftarrow \psi_i(v) \wedge S_i$
20: $\psi_i(\epsilon_i) \leftarrow \psi_i(\epsilon_i) \wedge g_0$
21: let $e_1, \ldots, e_m$ be a topologically ordered
sequence of all call-edges in $\mathcal{A}_i$
that can reach v
22: **for all** $e_k = (u, w) \in e_1, \ldots, e_m$ **do**
23: UPDATE($\mathcal{J}(e_k)$, GUARD(g_k), SUM(s_k))
24: **else**
25: **if** $i = 1$ **then**
26: Terminate with UNSAFE
27: $R^{\mathcal{C}} \leftarrow R^{\mathcal{C}} \setminus \{(\mathcal{A}_l, \mathcal{A}_i) \mid (\mathcal{A}_l, \mathcal{A}_i) \in R^{\mathcal{C}}\}$
28: **for all** $\{\mathcal{A}_j \mid (\mathcal{A}_j, \mathcal{A}_i) \in R^{\mathcal{J}}\}$ **do**
29: RESET($\mathcal{A}_j$)

Require: $\mathcal{A}_i$ is uncovered, safe, and complete
30: **function** UPDATEGUARD(ARG $\mathcal{A}_i$)
31: $G_i \leftarrow \psi(\epsilon_i)$

is not applicable indefinitely. Note that once an ARG is covered, all ARGs it covers are uncovered (line 5).

REFINEARG is the core of WHALE. Given an exit node v of some unsafe ARG $\mathcal{A}_i$, it checks whether there exists an interprocedural counterexample in $\mathcal{I}^{\mathcal{A}}(P)$, i.e., an interprocedural (ϵ_i, v)-path that satisfies the guard G_i and violates the summary S_i. This is done using IARGCOND to construct a condition *cond* that is satisfiable if and only if there is a counterexample (line 16). If *cond* is satisfiable and $i = 1$, then there is a counterexample to $\{G_1\}\, B_{F_1}\, \{S_1\}$, and WHALE terminates (line 24). If *cond* is satisfiable and $i \neq 1$, the guard and the summary of $\mathcal{A}_i$ are invalidated, all ARGs covered by $\mathcal{A}_i$ are uncovered, and all ARGs used to justify call-edges of $\mathcal{A}_i$ are reset (lines 25-26). If *cond* is unsatisfiable, then there is no counterexample in the current iARG. However, since the iARG represents only a partial unrolling of the program, this does not imply that the program is safe. In this case, REFINEARG uses interpolants to *guess* guards and summaries of functions called from $\mathcal{A}_i$ (lines 17-22) that can be used to replace their under-approximations without introducing new counterexamples.

The two primary distinctions between WHALE and IMPACT are in constructing a set of formulas to represent an ARG and in using interpolants to guess function summaries from these formulas. We describe these below.

6.5.1 Interprocedural ARG Condition

An *ARG condition* of an ARG $\mathcal{A}$ is a formula φ such that every satisfying assignment to φ corresponds to an execution through $\mathcal{A}$, and vice versa. A naive way to construct it is to take a disjunction of all the *path*

conditions of the paths in the ARG. ARG Conditions are the similar to DAG Conditions, DAGCond, in Chapter 3: they encode paths through a DAG labeled with formula-labeled edges. Here, we use ARG Conditions as building blocks for encoding interprocedural program paths. An *interprocedural ARG condition* of an ARG $\mathcal{A}$ in an iARG $\mathcal{I}^{\mathcal{A}}(P)$ (computed by the function iARGCond) is a formula φ whose every satisfying assignment corresponds to an interprocedural execution through $\mathcal{A}_i$ in $\mathcal{I}^{\mathcal{A}}(P)$ and vice versa.

We assume that $\mathcal{A}_i$ is in static single assignment (SSA) form [CFR+91] (i.e., every variable is assigned at most once on every path). iARGCond uses the function ARGCond to compute a ARG condition[2]:

$$\text{ARGCond}(\mathcal{A}_i, X) \equiv C \wedge D, \text{ where}$$
$$C = c_{\epsilon_i} \wedge \bigwedge_{v \in V_i'} \{c_v \Rightarrow \bigvee \{c_w \mid (v,w) \in E_i\}\},$$
$$D = \bigwedge_{(v,w) \in E_i'} \{(c_v \wedge c_w) \Rightarrow [\![\tau_i(v,w)]\!] \mid \tau_i(v,w) \text{ is simple}\}, \quad (6.1)$$

c_i are Boolean *control variables* for nodes of $\mathcal{A}_i$ such that a variable c_v corresponds to node v, and $V_i' \subseteq V_i$ and $E_i' \subseteq E_i$ are sets of nodes and edges, respectively, that can reach a node in the set of exit nodes X. Intuitively, C and D encode all paths through $\mathcal{A}_i$ and the corresponding path condition, respectively. ARGCond ignores call statements which (in SSA) corresponds to replacing calls by non-deterministic assignments.

Example 6.1. *Consider computing* $\text{ARGCond}(\mathtt{A}_1', \{8, 8a\})$ *for the ARG* $\mathtt{A}_1'$ *in Figure 6.1, where* c_8 *and* c_{8a} *represent the two exit nodes, on the left and on the right, respectively. Then,*

$$\begin{aligned} C =& c_2 \wedge (c_2 \Rightarrow (c_3 \vee c_5)) \wedge (c_3 \Rightarrow c_8) \wedge \\ & (c_5 \Rightarrow c_6) \wedge (c_6 \Rightarrow c_7) \wedge (c_7 \Rightarrow c_{8a}) \text{ and} \\ D =& (c_2 \wedge c_3 \Rightarrow p \leqslant 100) \wedge \\ & (c_3 \wedge c_8 \Rightarrow r = p - 10) \wedge \\ & (c_2 \wedge c_5 \Rightarrow p \leqslant 100) \wedge \\ & (c_5 \wedge c_6 \Rightarrow p_1 = p + 11). \end{aligned}$$

Any satisfying assignment to $C \wedge D$ *represents an execution through the path 2,3,8 or 2,5,...,8, where the call statements on edges (6,7) and (7,8) set* p_2 *and* r *non-deterministically.*

The function $\text{iARGCond}(\mathcal{A}_i, X)$ computes an interprocedural ARG condition for a given ARG and a set X of exit nodes of $\mathcal{A}_i$ by using ARGCond and interpreting function calls. A naive encoding is to inline every call-edge e with the justifying ARG $\mathcal{J}(e)$, but this results in a monolithic formula which hinders interpolation in the next step of RefineARG. Instead, we define it as follows:

$$\text{iARGCond}(\mathcal{A}_i, X) \equiv \text{ARGCond}(\mathcal{A}_i, X) \wedge \bigwedge_{k=1}^{m} \mu_k, \text{ where}$$
$$\mu_k \equiv (c_{v_k} \wedge c_{w_k}) \Rightarrow ((\vec{p}_{\sigma(j)}, \vec{r}_{\sigma(j)} = \vec{a}, \vec{b}) \wedge \text{iARGCond}(\mathcal{A}_j, V_j^e)), \quad (6.2)$$

[2] In practice, we use a more efficient encoding described in [GCS11].

m is the number of call-edges in $\mathcal{A}_i$, $e = (v_k, w_k)$ is the kth call-edge[3], $\mathcal{A}_j = \mathcal{J}(e)$, and $\tau(e)$ is $\vec{b} = F_{\sigma(j)}(\vec{a})$. Intuitively, μ_k is the under-approximation of the kth call-edge e in $\mathcal{A}_i$ by the traces in the justifying ARG $\mathcal{A}_j = \mathcal{J}(e)$. Note that iARGCond always terminates since the justification relation is acyclic.

Example 6.2. *Following Example 6.1,* $\text{iARGCond}(\mathtt{A}'_1, \{8, 8a\})$ *is* $(C \wedge D) \wedge \mu_1 \wedge \mu_2$*, where* $C \wedge D$ *are as previously defined, and* μ_1, μ_2 *represent constraints on the edges* $(6, 7)$ *and* $(7, 8)$*. Here,*

$$\mu_1 = (c_6 \wedge c_7) \Rightarrow ((p' = p_1 \wedge p_2 = r') \wedge \text{ARGCond}(\mathtt{A}_2, \{8\})).$$

That is, if an execution goes through the edge (6,7), then it has to go through the paths of $\mathtt{A}_2$*—the ARG justifying this edge. Using primed variables avoids name clashes between the locals of the caller and the callee.*

Lemma 6.1. *Given an iARG* $\mathcal{I}^{\mathcal{A}}(P)$*, an ARG* $\mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P)$*, and a set of exit nodes* X*, there exists a total onto map from satisfying assignments of* $\text{iARGCond}(\mathcal{A}_i, X)$ *to interprocedural* (ϵ_i, X)*-executions in* $\mathcal{I}^{\mathcal{A}}(P)$*.*

A corollary to Lemma 6.1 is that for any pair of formulas G and S, $G \wedge \text{iARGCond}(\mathcal{A}_i, X) \wedge S$ is unsatisfiable if and only if there does not exist an execution in $\mathcal{A}_i$ that starts at ϵ_i in a state satisfying G and ends in a state $v \in X$ satisfying S.

6.5.2 Guessing Guards/Summaries with State/Transition Interpolants

Our goal now is to show how under-approximations of callees in formulas produced by iARGCond can be generalized. First, we define a function

$$\text{SpecCond}(\mathcal{A}_i, X, I) \equiv \text{ARGCond}(\mathcal{A}_i, X) \wedge \bigwedge_{k=1}^{m} \mu_k, \text{ where}$$

$$\mu_k = (c_{v_k} \wedge c_{w_k}) \Rightarrow ((\vec{p}_{\sigma(j)}, \vec{r}_{\sigma(j)} = \vec{a}, \vec{b}) \wedge (q_k \Rightarrow t_k)),$$

$I = \{(q_k, t_k)\}_{k=1}^{m}$ is a sequence of formulas over program variables, and the rest is as in the definition of iARGCond. SpecCond is similar to iARGCond, except that it takes a sequence of pairs of formulas (pre- and postconditions) that act as specifications of the called functions on the call-edges $\{e_k\}_{k=1}^{m}$ along the paths to X in $\mathcal{A}_i$. Every satisfying assignment of $\text{SpecCond}(\mathcal{A}_i, X, I)$ corresponds to an execution through $\mathcal{A}_i$ ending in X, where each call-edge e_k is interpreted as $\texttt{assume}(q_k \Rightarrow t_k)$.

Lemma 6.2. *Given an iARG* $\mathcal{I}^{\mathcal{A}}(P)$*, an ARG* $\mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P)$*, a set of exit nodes* X*, and a sequence of formulas* $I = \{(q_k, t_k)\}_{k=1}^{m}$*, there exists a total and onto map from satisfying assignments of* $\text{SpecCond}(\mathcal{A}_i, X, I)$ *to* (ϵ_i, X)*-executions in* $\mathcal{A}_i$*, where each call-edge* e_k *is interpreted as* $\texttt{assume}(q_k \Rightarrow t_k)$*.*

Given an unsatisfiable formula $\Phi = G_i \wedge \text{iARGCond}(\mathcal{A}_i, X) \wedge \neg S_i$, the goal is to find a sequence of pairs of formulas $I = \{(q_k, t_k)\}_k$ such that $G_i \wedge \text{SpecCond}(\mathcal{A}_i, X, I) \wedge \neg S_i$ is unsatisfiable, and for every t_k, $\text{iARGCond}(\mathcal{A}_j, V_j^e) \Rightarrow t_k$, where $\mathcal{A}_j = \mathcal{J}(e_k)$. That is, we want to *weaken* the under-approximations of callees in Φ, while keeping Φ unsatisfiable. For this, we use interpolants.

We require a stronger notion of interpolants than usual:

[3]Recall, call-edges are ordered in some linearization of a topological order of RG U_i.

Definition 6.1 (State/Transition Interpolants)**.** Let $\Pi = \varphi_0 \wedge \cdots \wedge \varphi_{n+1}$ be unsatisfiable. A sequence of formulas $g_0, s_0, \ldots, g_{n-1}, s_{n-1}, g_n$ is a *state/transition interpolant sequence* of Π, written $\text{STItp}(\Pi)$, if and only if

1. $\varphi_0 \Rightarrow g_0$,
2. $\forall i \in [0, n] \cdot \varphi_{i+1} \Rightarrow s_i$,
3. $\forall i \in [0, n] \cdot (g_i \wedge s_i) \Rightarrow g_{i+1}$,
4. and $g_n \wedge \varphi_{n+1}$ is unsatisfiable.

We call g_i and s_i the state- and transition-interpolants, respectively. □

$\text{STItp}(\Pi)$ can be computed by a repeated application of current SMT interpolation algorithms [CGS10] on the same resolution proof:

$$g_i = \text{Itp}(\bigwedge_{j=0}^{i} \varphi_j, \bigwedge_{j=i+1}^{n+1} \varphi_j, pf) \text{ and}$$

$$s_i = \text{Itp}(\varphi_i, \bigwedge_{j=0}^{i-1} \varphi_j \wedge \bigwedge_{j=i+1}^{n+1} \varphi_j, pf),$$

where pf is a fixed resolution proof and $\text{Itp}(A, B, pf)$ is a Craig interpolant of (A, B) from pf.

Recall that RefineARG (Algorithm 7), on line 16, computes a formula $cond = G_i \wedge \varphi \wedge \bigwedge_{k=1}^{m} \mu_k \wedge \neg S_i$ using iARGCond for ARG $\mathcal{A}_i$ and an exit node v, where μ_k is an under-approximation representing the call-edge $e_k = (u_k, w_k)$. For simplicity of presentation, let $\tau(e_k)$ be $\vec{b}_k = F_k(\vec{a}_k)$. Assume $cond$ is unsatisfiable and let $g_0, s_0, \ldots, s_m, g_{m+1}$ be state/transition interpolants for $cond$. By definition, each s_k is an over-approximation of μ_k that keeps $cond$ unsatisfiable. Similarly, g_0 is an over-approximation of G_i that keeps $cond$ unsatisfiable, and g_k, where $k \neq 0$, is an over-approximation of the executions of $\mathcal{A}_i$ assuming that all call statements on edges $e_k, \ldots, e_m$ are non-deterministic. This is due to the fact that $(G_i \wedge \varphi \wedge \mu_1 \wedge \cdots \wedge \mu_{j-1}) \Rightarrow g_j$. Note that $g_0, s_0, \ldots, s_m, g_{m+1}$ are also state/transition interpolants for the formula

$$G_i \wedge \varphi \wedge (g_1 \Rightarrow s_1) \wedge \cdots \wedge (g_m \Rightarrow s_m) \wedge \neg S_i.$$

The goal (lines 18–22) is to use the sequence $\{(g_k, s_k)\}_{k=1}^{m}$ to compute a sequence $I = \{(q_k, t_k)\}_{k=1}^{m}$ such that

$$G_i \wedge \text{SpecCond}(\mathcal{A}_i, \{v\}, I) \wedge \neg S_i$$

is unsatisfiable. By definition of an interpolant, s_k is over the variables $\vec{a}_k$, $\vec{b}_k$, c_{u_k}, and c_{w_k}, whereas t_k has to be over $\vec{p}_k$ and $\vec{r}_k$, to represent a summary of F_k. Similarly, g_k is over $\vec{a}_k$, $\vec{b}_k$, c_{u_j}, and c_{w_j} for all $j \geqslant k$, whereas q_k has to be over $\vec{p}_k$ to represent a guard on the calling contexts. This transformation is done using the following functions:

$$\text{Sum}(s_k) \equiv s_k[c_{u_k}, c_{w_k} \leftarrow \top][\vec{a}_k, \vec{b}_k \leftarrow \vec{p}_k, \vec{r}_k]$$
$$\text{Guard}(g_k) \equiv \exists Q \cdot g_k[c_u \leftarrow (u_k \sqsubseteq u) \mid u \in V_i][\vec{a}_k \leftarrow \vec{p}_k],$$

where the notation $\varphi[x \leftarrow y]$ stands for a formula φ with all occurrences of x replaced by y, $w \sqsubseteq u$ means that a node u is reachable from w in $\mathcal{A}_i$, and Q is the set of all variables in g_k except for $\vec{a}_k$.

Given a transition interpolant s_k, $\textsc{Sum}(s_k)$ is an over-approximation of the set of reachable states by the paths in $\mathcal{J}(u_k, w_k)$. $\textsc{Guard}(g_k)$ sets all (and only) successor nodes of u_k to true, thus restricting g_k to executions reaching the call-edge (u_k, w_k); furthermore, all variables except for the arguments $\vec{a}_k$ are existentially quantified, effectively over-approximating the set of parameter values with which the call on (u_k, w_k) is made.

Lemma 6.3. *Given an ARG $\mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P)$, and a set of exit nodes X, let $\Phi = G_i \wedge \textsc{iARGCond}(\mathcal{A}_i, X) \wedge \neg S_i$ be unsatisfiable and let $g_0, s_0, \ldots, s_m, g_{m+1}$ be $\textsc{StItp}(\Phi)$. Then,*

$$G_i \wedge \textsc{SpecCond}(\mathcal{A}_i, X, \{(\textsc{Guard}(g_k), \textsc{Sum}(s_k))\}_{k=1}^{m}) \wedge \neg S_i$$

is unsatisfiable.

Example 6.3. *Let $cond = true \wedge \varphi \wedge \mu_1 \wedge \mu_2 \wedge (r < 91)$, where true is the guard of $\mathtt{A}'_1$, φ is $C \wedge D$ from Example 6.1, μ_1 and μ_2 are as defined in Example 6.2, and $(r < 91)$ is the negation of the summary of $\mathtt{A}'_1$. A possible sequence of state/transition interpolants for* cond *is $g_0, s_0, g_1, s_1, g_2, s_2, g_3$, where*

$$\begin{aligned} g_1 &= (r < 91 \Rightarrow (c_6 \wedge c_7 \wedge c_{8a})), \\ s_1 &= ((c_6 \wedge c_7) \Rightarrow p_2 \geqslant 91), \\ g_2 &= (r < 91 \Rightarrow (c_7 \wedge c_{8a} \wedge p_2 \geqslant 91)), \text{ and} \\ s_2 &= ((c_7 \wedge c_{8a}) \Rightarrow r \geqslant 91). \end{aligned}$$

Hence, $\textsc{Guard}(g_1) = \exists r \cdot r < 91$ (since all c_u, where node u is reachable from node 6, are set to true), $\textsc{Sum}(s_1) = r \geqslant 91$ (since r is the return variable of `mc91`), $\textsc{Guard}(g_2) = p \geqslant 91$, and $\textsc{Sum}(s_2) = r \geqslant 91$.

RefineARG uses $(\textsc{Guard}(g_k), \textsc{Sum}(s_k))$ of each edge e_k to strengthen the guard and summary of its justifying ARG $\mathcal{J}(e_k)$. While $\textsc{Guard}(g_k)$ may have existential quantifiers, it is not a problem for iARGCond since existentials can be skolemized. However, its may be a problem for deciding the precondition of CoverArg. In practice, we eliminate existentials using interpolants by observing that for a complete ARG $\mathcal{A}_i$, $\psi_i(\epsilon_i)$ is a quantifier-free safe over-approximation of the guard. Once an ARG $\mathcal{A}_i$ is complete, UpdateGuard in Algorithm 7 is used to update G_i with its quantifier-free over-approximation. Hence, an expensive quantifier elimination step is avoided.

6.5.3 Soundness and Completeness

By Lemma 6.1 and Lemma 6.2, Whale maintains an invariant that every complete, safe and uncovered ARG $\mathcal{A}_i$ means that its corresponding function satisfies its guard and summary *assuming* that all other functions satisfy the corresponding guards and summaries of all ARGs in the current iARG. Formally, let Y and Z be two sets of triples defined as follows:

$$\begin{aligned} Y &\equiv \{\{G_j\}\, \vec{b} = F_{\sigma(j)}\, (\vec{a})\{S_j\} \mid \mathcal{A}_j \in \mathcal{I}^{\mathcal{A}}(P) \text{ is uncovered or directly covered}\} \\ Z &\equiv \{\{G_i\}\, B_{F_{\sigma(i)}}\, \{S_i\} \mid \mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P) \text{ is safe, complete, and uncovered}\} \end{aligned}$$

Whale maintains the invariant $Y \vdash Z$. Furthermore, if the algorithm terminates, every uncovered ARG is safe and complete, and every directly covered ARG is justified by an uncovered one. This satisfies the premise of Hoare's (generalized) proof rule for mutual recursion and establishes soundness of Whale.

	Whale			Wolverine 0.5	Blast 2.5			
Program	**#ARGs**	**#Refine**	**Time**	**Time**	**Time (B1)**	**Time (B2)**	**#Preds (B1)**	**#Preds (B2)**
`ddd1.c`	5	3	**0.43**	4.01	4.64	1.71	15	8
`ddd2.c`	5	3	**0.59**	5.71	5.29	2.65	16	10
`ddd3.c`	6	5	**20.19**	30.56	48	20.32	25	16
`ddd1err.c`	5	1	**0.16**	3.82	0.42	1.00	25	8
`ddd2err.c`	5	1	**0.28**	5.72	0.44	0.96	5	8
`ddd3err.c`	5	11	126.4	**17.25**	TO	43.11	TO	37
`ddd4err.c`	6	1	5.73	**1.76**	24.51	CR	19	CR

Table 6.1: Evaluation of Whale: results and tool comparison.

Whale is complete for Boolean programs, under the restriction that the three main operations are scheduled fairly (specifically, CoverARG is applied infinitely often). The key is that Whale only uses interpolants over program variables in a current scope. For Boolean programs, this bounds the number of available interpolants. Therefore, all incomplete ARGs are eventually covered.

Theorem 6.1. Whale *is sound. Under fair scheduling, it is complete for Boolean programs.*

6.6 Experimental Evaluation

We implemented Whale in an early experimental version of the UFO_{app} tool. The development, implementation, and evaluation of Whale preceded that of the Ufo and Vinta algorithms presented in Chapters 4 and 5, respectively. For satisfiability checking and interpolant generation, we use the MathSAT4 SMT solver [BCF+08].

Our implementation of Whale is a particular heuristic determinization of the three operations described in Section 6.5: A FIFO queue is used to schedule the processing of ARGs. Initially, the queue contains only the main ARG $\mathcal{A}_1$. When an ARG is picked up from the queue, we first try to cover it with another ARG, using CoverARG. In case it is still uncovered, we apply UpdateARG and RefineARG until they are no longer applicable, or until RefineARG returns a counterexample. Every ARG created by UpdateARG or modified by Reset is added to the processing queue. Furthermore, we use several optimizations not reported here. In particular, we merge ARGs of same the function. The figures reported in this section are for the number of combined ARGs and do not represent the number of function calls considered by the analysis.

Our goal in evaluating Whale is two-fold: (1) to compare effectiveness of our interpolation-based approach against traditional predicate abstraction techniques, and (2) to compare our interprocedural analysis against intraprocedural interpolation-based algorithms. For (1), we compared Whale with Blast [BHJM07]. For (2), we compared Whale with Wolverine [KW11], a recent software model checker that implements Impact algorithm [McM06] (it inlines functions and, thus, does not handle recursion).

For both evaluations, we used non-recursive low-level C programs written for the pacemaker grand challenge [pac]. Pacemakers are devices implanted in a human's body to monitor heart rate and send electrical signals (paces) to the heart when required. We wrote test harnesses to simulate the pacemaker's interaction with the heart on one of the most complex pacemaker operation modes (DDD). The major actions of a pacemaker are sensing and pacing. Periodically, a pacemaker suspends its sensing operation and then turns it back on. The properties we checked involved verifying correct sequences of toggling

sensing operations, e.g., that sensing is not suspended for more than two time steps, where we measured time steps by the number of interrupts the pacemaker receives.

Table 6.1 summarizes the results of our experiments. Blast was run in two configurations, B1 and B2.[4] Wolverine was run in its default (optimal) configuration. For Whale, we show the number of ARGs created and the number of calls to RefineARG for each program. For Blast, we show the number of predicates needed to prove or refute the property in question. 'CR' and 'TO' denote a crash and an execution taking longer than 180s, respectively. The programs named `ddd`*i*`.c` are safe; `ddd`*i*`err.c` have errors. While all programs are small (∼300 LOC), their control structure is relatively complex.

For example, Table 6.1 shows that Whale created five ARGs while processing `ddd3.c`, called RefineARG three times and proved the program's correctness in 0.59 seconds. Blast's configuration B1 tool 5.29 seconds and used 16 predicates, whereas B2 took 2.65 seconds and used 10 predicates. Wolverine's performance was comparable to B1, verifying the program in 5.71 seconds.

For most properties and programs, we observe that Whale outperforms Wolverine and Blast (in both configurations). Note that neither of the used Blast configurations could handle the entire set of programs without crashing or timing out. `ddd3err.c` contains a deep error, and to find it, Whale spends a considerable amount of time in SMT solver calls, refining and finding counterexamples to a summary, until the under-approximation leading to the error state is found. For this particular example, we believe Wolverine's dominance is an artifact of its search strategy. In the future, we want to experiment with heuristics for picking initial under-approximations and heuristics for refining them, in order to achieve faster convergence.

6.7 Related Work

In this section, we place Whale within the landscape of related work.

At a high level, all interpolation-based verification techniques examine a bounded version (an under-approximation) of a given program and hypothesize a proof of correctness for the whole program. The Whale algorithm is no different; what is unique is the modular nature of the computed proof. Specifically, Whale uses interpolants to hypothesize a procedure-modular proof, where the behaviour of each procedure is defined by pre- and post-conditions. Whale can be viewed as a reformulation of McMillan's LAWI [McM06] algorithm to programs with procedures and recursion. LAWI unrolls the control-flow graph of the program into a tree, and labels nodes of the tree with over-approximations of reachable states at their respective locations. Whale, on the other hand, unrolls the call graph of the program into a tree, and labels nodes of the tree, which map to procedures, with procedure summaries. Our notion of ARG covering is analogous to McMillan's vertex covering lifted to the ARG level. While LAWI unrolls loops until all vertices are covered or fully expanded (thus, an invariant is found), Whale unrolls recursive calls until all ARGs are covered or fully expanded (completed). One advantage of Whale is that it encodes all intraprocedural paths by a single SMT formula. Effectively, this results in delegating intraprocedural covering to the SMT solver.

Whale can also be viewed as an extension of our DAG-interpolation-based algorithm, Ufo, presented in Chapters 3 and 4, to the interprocedural setting. Specifically, Whale extends Ufo by constructing a forest of ARGs for multiple procedures—instead of a single ARG. Ufo uses DAG interpolants to label nodes of an ARG. Whale uses state/transition interpolants to label whole ARGs, without labeling

[4] B1 is `-dfs -craig 2 -predH 0` and B2 is `-msvc -nofp -dfs -tproj -cldepth 1 -predH 6 -scope -nolattice`.

nodes of an ARG.

In [McM10], interpolants are used as blocking conditions on infeasible symbolic execution paths and as means of computing function summaries. This approach differs from Whale in that the exploration is not property-driven and thus is more suited for bug finding than verification. Also, handling unbounded loops and recursion requires addition of auxiliary variables into the program. In [HHP10], Heizmann et al. present an interprocedural verification technique that uses interpolants to generalize an infeasible interprocedural path to a set of infeasible paths. Automata-based language inclusion is then used to check if all program paths are included in the infeasible set. Whale computes modular proofs of correctness, a summary per procedure, whereas [HHP10]'s notion of a proof is monolithic—a nested word automata that includes every possible execution.

McMillan and Rybalchenko [MR13] proposed a similar extension of LAWI to procedures and recursion, called Duality. In a similar fashion to Whale, Duality unrolls the call graph of the program and uses interpolants to compute procedure summaries. Duality uses a *deeper* form of interpolants, called *tree interpolants*, to label an iARG-like structure. From an encoding of the iARG, tree interpolants enable generating a well-labeling for the whole iARG. In contrast, Whale uses state/transition interpolants, which only produce labels for the children of the root of the ARG, and has to proceed downwards to label the rest. A thorough technical comparison of Duality's approach with Whale is given in [MR13]. Rümmer et al. [RHK13b] generalized tree and transition interpolants into *disjunctive interpolants*. Specifically, Rümmer et al. recognized that we can take an interpolant between a formula and one of its positively appearing sub-formulas. In the context of interprocedural verification, given a formula encoding an unrolling of the program with sub-formulas encoding under-approximations of function calls, disjunctive interpolants can be used to generalize the under-approximations. Transition interpolants are a restricted form of disjunctive interpolants where sub-formulas can only appear as outer-level conjuncts.

Synergy [GHK+06] and its interprocedural successor Smash [GNRT10] start with an approximate partitioning of reachable states of a given program. Partition refinement is guided by the weakest precondition computations over infeasible program paths. The main differences between Whale and [GHK+06, GNRT10] are: (a) interpolants focus on relevant facts and can force faster convergence than weakest preconditions [HJMM04, McM10]; (b) our use of interpolants does not require an expensive quantifier elimination step employed by Smash to produce summaries; and (c) Synergy and Smash use concrete test cases to guide their choice of program paths to explore. Compared to Whale, this makes them better suited for bug finding.

Abstraction-based techniques, employing a forward abstract transformer, as implemented in Slam [BR01] and others, typically use classical summary-based interprocedural data-flow analysis algorithms to compute inductive invariants for programs with procedures [SP81, RHS95]. In cases where the number of possible configurations (states of procedures) is finite, e.g., in Boolean programs, techniques employing [RHS95] are polynomial in the number of reachable configurations. In contrast, IB techniques like Whale and Duality unroll the call graph into a tree in order to produce an SMT encoding, risking an avoidable exponential explosion.

6.8 Conclusion

In this section, we summarize this chapter and compare and contrast intraprocedural and interprocedural analyses.

Summary Interpolation-based verification has received a considerable amount of attention in recent years for a number of reasons: its lightweight nature (no abstract domain used), the rise of SMT solvers, and the ability of interpolants to focus on relevant facts for the property at hand and compute simple invariants. Despite the great deal of research in the area, very little work has addressed the problem of interprocedural analysis with interpolants.

In this chapter, we presented a new IB technique for verifying safety of programs with procedures and recursion. Our algorithm, Whale, lifts the lazy abstraction with interpolants [McM06] algorithm to the granularity of procedures. Whale thus unrolls the call graph of a given program—instead of the control-flow graph—and computes procedure-modular proofs in the form of procedure summaries. Whale is enabled by state/transition interpolants, a new notion of interpolants that allow over-approximating bounded procedure unrollings as procedure summaries.

Interprocedural vs. Intraprocedural Algorithms We have presented in this dissertation intraprocedural and interprocedural verification algorithms. This raises the question of why do we need intraprocedural algorithms in the presence of more general interprocedural ones; or even the opposite question: why do we need interprocedural algorithms when all programs are, after compilation, intraprocedural in nature. We discuss these points below.

- Given an interprocedural analysis, like the one presented in this chapter, we can always convert all loops in a program with a single procedure into recursive procedures and use the interprocedural algorithm. Theoretically speaking, for a number of program analysis questions, the intraprocedural problem is (likely) *easier* than the interprocedural one [Rep96]. Specifically, Reps showed that for a class data-flow analysis problems, interprocedural analysis is P-hard, meaning that it is unlikely to have parallel, NC, algorithms unless P=NC. On the other hand, intraprocedural analysis reduces to a graph-reachability problem with NC algorithms. Thus, it is likely that interprocedural analysis is harder, unless all polynomial problems are *efficiently parallelizable* (in NC).

- Alternatively, why can't we convert a program with procedures and recursion into a program with a single procedure and use intraprocedural analyses? First, suppose the program has no recursion. The conversion is now straightforward: we simply inline all procedures into their callers. The problem here is this might result in a single-procedure program that is *exponentially larger* than the original program with procedures. Consider the simple example where the main procedure F_1 calls procedure F_2 at n call sites and procedure F_2 calls F_3 also n times. Then, after inlining, F_1 will have n^2 copies of F_3. Extending this example to longer procedure-call chains easily shows that leaf procedures in the call graph can get copied exponentially many times in the size of the call graph.

  Now suppose that the program has recursion. We can convert it into a program with a single procedure by explicitly managing the call stack. This, of course, can easily result in a very complex program that is hard to analyze intraprocedurally. For instance, we now need to model the stack in our logic and abstract domains, a difficult problem with no well-developed techniques.

At a more practical level, procedure summarization can be helpful when we are analyzing frequently called procedures, where summaries can be stored and reused, e.g., for verifying other programs calling the same library of procedures. Inlining would result in a monolithic program, making it non-trivial to reuse computed invariants/summaries. Furthermore, developing efficient intraprocedural analyses can directly benefit interprocedural analysis, as demonstrated by a number of recent works [GNRT10, AKNR12, YFCW14].

Chapter 7

Tool Support: Architecture and Engineering

"Numbers don't lie, check the scoreboard."
— Shawn Corey Carter

7.1 Introduction

Chapters 3-6 introduced new verification algorithms and concepts and demonstrated their practical utility. In the process of experimenting with and evaluating our algorithms, we designed, implemented, and refined a verification tool and framework called UFO_{app}, which we have used for verifying (and finding bugs in) sequential C programs.

In this chapter, we describe the architecture of UFO_{app}, its prominent optimizations, and its success in the International Software Verification Competition (SV-COMP). The main features of UFO_{app} are:

- UFO_{app} is a framework for building and experimenting with IB, AB, and combined IB/AB verification algorithms. It is *parameterized* by the abstract post operator, refinement strategy, and expansion strategy. The architecture and parameterization of UFO_{app} and the underlying LLVM framework provide users with an extensible environment for experimenting with different software verification algorithms.

- The UFO_{app} framework contains a number of novel instantiations described in Chapters 3-5: purely interpolation-based algorithm, combined IB/AB algorithms with numerous abstract domains, and purely AB algorithms. At the core of UFO_{app} is an efficient DAG interpolation procedure for hypothesizing invariants and strengthening results of abstract interpretation. Unlike other tools that enumerate paths explicitly, for example, [McM06, KW11, BHJM07], UFO_{app} delegates path enumeration to an efficient SMT solver.

- UFO_{app} is implemented using the very popular, open source LLVM compiler infrastructure [LA04]. Since LLVM is a well-maintained, well-documented, and continuously evolving framework, it allows UFO_{app} users to easily integrate program analyses, transformations, and other tools built on LLVM

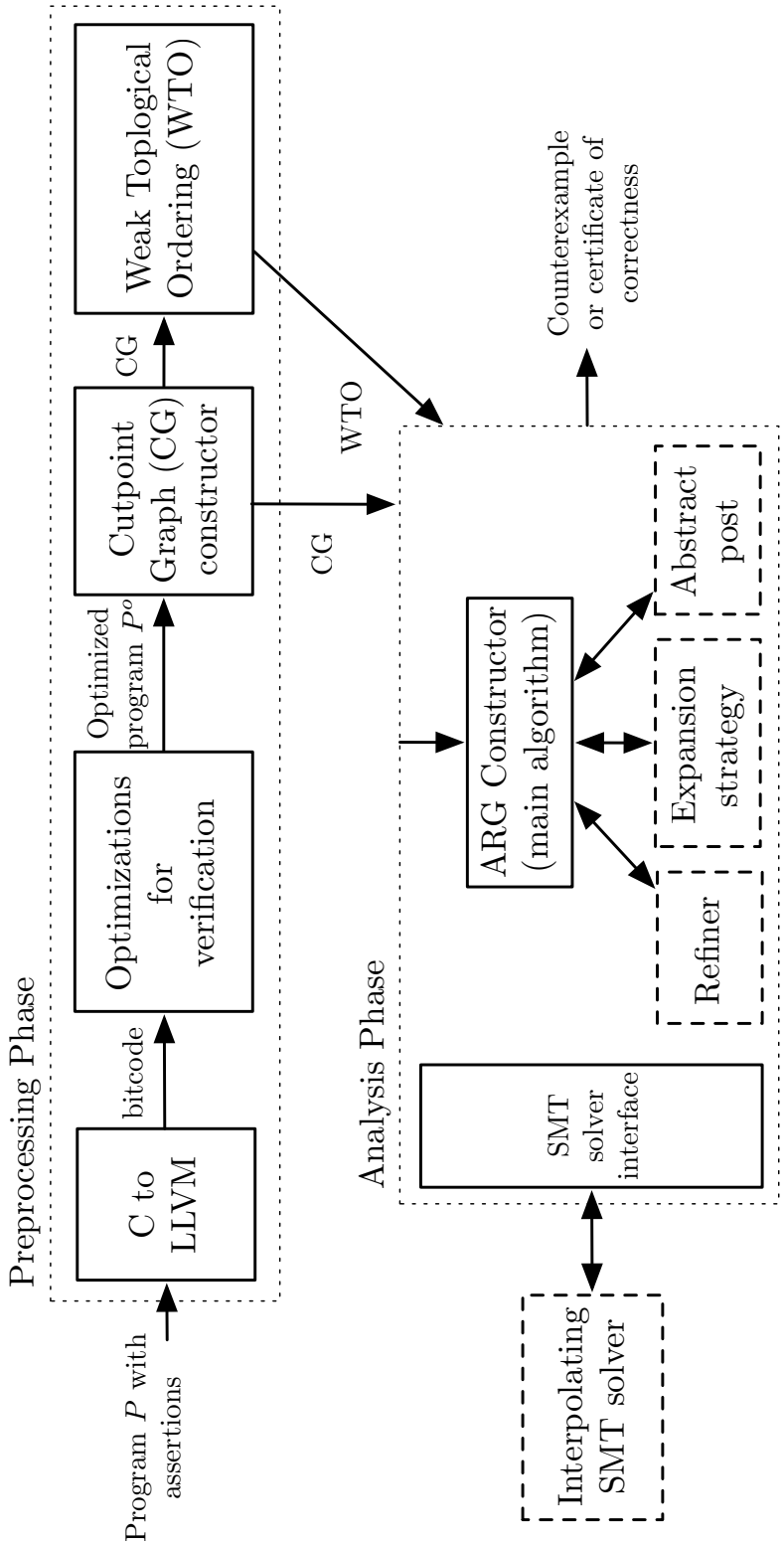


Figure 7.1: Architecture of UFO_{app}.

(e.g., Klee [CDE08]) as they become available. Furthermore, since UFO$_{app}$ analyzes LLVM bitcode (LLVM's intermediate language), it is possible to experiment with verifying programs written in other languages compilable to LLVM bitcode, such as C++, Ada, and Swift.

UFO$_{app}$'s source code and compilation instructions are available at `bitbucket.org/arieg/ufo`.

Organization

This chapter is organized as follows:

- In Section 7.2, we described the architecture and implementation of UFO$_{app}$.
- In Section 7.3, we describe the important optimizations implemented in the UFO$_{app}$ framework.
- In Section 7.4, we discuss UFO$_{app}$'s participation in SV-COMP and its parallel (multiple configurations) approach.
- Finally, in Section 7.5, we conclude this chapter.

7.2 Architecture and Implementation

UFO$_{app}$ is implemented on top of the LLVM compiler infrastructure [LA04]. See Figure 7.1 for an architectural overview. UFO$_{app}$ accepts as input a C program P with assertions. For simplicity of presentation, let $P = (V, T, \phi_{\mathcal{I}}, \phi_{\mathcal{E}})$, where V is the set of program variables, T is the transition relation of the program (over V and V', the set of primed variables), $\phi_{\mathcal{I}}$ is a formula describing the set of initial states, and $\phi_{\mathcal{E}}$ is a formula describing the set of error states.

First, P goes through a *preprocessing phase* where it is compiled into LLVM bitcode (intermediate representation) and optimized and transformed for verification purposes, resulting in a semantically equivalent but optimized program $P^o = (V^o, T^o, \phi^o_{\mathcal{I}}, \phi^o_{\mathcal{E}})$.

Then, the *analysis phase* verifies P^o and either outputs a certificate of correctness or a counterexample. A certificate of correctness for P^o is a safe inductive invariant I such that (1) $\phi^o_{\mathcal{I}} \Rightarrow I$, (2) $I \wedge T^o \Rightarrow I'$, and (3) $I \wedge \phi^o_{\mathcal{E}}$ is unsatisfiable.

7.2.1 Preprocessing Phase

We now describe the various components of the preprocessing phase.

C to LLVM The first step converts the program P to LLVM bitcode using the `llvm-gcc` or `clang` compilers [cla]. LLVM bitcode is the LLVM infrastructure's intermediate representation (IR) on which all analyses and transformations are performed.

Optimizations for Verification A number of native LLVM optimizations are then applied to the bitcode, the most important of which are *function inlining* (`inline`) and SSA conversion (`mem2reg`). Since UFO$_{app}$ implements an intraprocedural analysis, it requires all functions to be inlined into `main`. In order to exploit efficient SMT program encoding techniques like [GCS11], UFO$_{app}$ expects the program to be in SSA form; SSA form allows us to encode straight-line programs without requiring frame conditions explicitly specifying that variables did not change. A number of standard program simplifications are

also performed at this stage, with the goal of simplifying verification. The final result is the optimized program P^o. Mapping counterexamples from P^o back to the original C program P is made possible by the debugging information inserted into the generated bitcode by `clang`.

Before the above optimizations could be applied, we had to bridge the gap between the semantics of C assumed by LLVM (built for compiler construction) and the verification tasks/benchmarks. Consider, for example, the following snippet of C code:

```
int main(){
  int x;
  if (x == 0)
    func1();
  if (x != 0)
    func2();
  return 1;
}
```

After LLVM optimizations, this program is reduced to the *empty* program:

```
return 1;
```

LLVM replaces undefined values by constants that result in the simplest possible program. In our example, the conditions of both `if`-statements are assigned to `0`, even though they contradict each other. (C semantics provide compilers the freedom to transform code with undefined behaviours [WCC+12, WZKSL13]—in this case, the value of `x` is undefined.) On the other hand, verification benchmarks such as [Bey12] assume that without an explicit initialization the value of `x` is non-deterministic. To account for such semantic differences, a UFO_{app}-specific LLVM transformation is scheduled before optimizations are run. It initializes each variable with a call to an external function `nondet()`, forcing LLVM not to make assumptions about its value.

Cutpoint Graph and Weak Topological Ordering A cutpoint graph (CPG) is a "summarized" control-flow graph (CFG), where each node represents a cutpoint (loop head) and each edge represents a loop-free path through the CFG. Computed using the technique presented in [GCS11], the CPG is used as the main representation of the program P^o. Using it allows us to perform abstract post operations on loop-free segments, utilizing the SMT solver (e.g., in the case of predicate abstraction) for enumerating a potentially exponential number of paths. A weak topological ordering (WTO), see Section **??**, is an ordering of the nodes of the CPG that enables exploring it with a recursive iteration strategy starting with the inner-most loops and ending with the outer-most ones. CPG and WTO are computed on the program P^o and are implemented as LLVM passes (analyses).

7.2.2 Analysis Phase

The analysis phase, which receives the CPG and the WTO of P^o from the preprocessing phase, is comprised of the following components:

ARG Constructor The *ARG Constructor* is the main driver of the analysis. It maintains an abstract reachability graph (ARG) of the CPG annotated with formulas representing over-approximations of reachable states at each cutpoint. When the algorithm terminates without finding a counterexample, the annotated ARG represents a certificate of correctness in the form of a safe inductive invariant I for P^o. To compute annotations for the ARG, the ARG constructor uses three *parameterized components*:

- the abstract post, to annotate the ARG as it is being expanded;
- the refiner, to compute annotations that eliminate spurious counterexamples; and
- the expansion strategy, to decide where to restart expanding the ARG after refinement.

Abstract Post. The abstract post component takes a CPG edge and a formula ϕ_{pre} describing a set of states, and returns a formula ϕ_{post} over-approximating the states reachable from ϕ_{pre} after executing the CPG edge. UFO_{app} includes a number of abstract domains for computing abstract post: Boolean predicate abstraction, Cartesian predicate abstraction, intervals (Box) , intervals with disjunctions (Boxes), the family of *template constraint matrix* (TCM) domains [SSM05], as well as combinations of those.

Refiner. The refiner receives the current ARG with potential paths to an error location (i.e., the error location is not annotated with *false*). Its goal is either to find a new annotation for the ARG such that the error location is annotated with *false*, or to report a counterexample. UFO_{app} includes a number of implementations of refinement using (restricted) DAG interpolants.

Expansion Strategy. After the refinement, the ARG constructor needs to decide where to restart expanding the ARG. The *expansion strategy* specifies this parameter. UFO_{app} includes an eager strategy and a lazy strategy. The lazy strategy is utilized by Ufo and Vinta to continue expanding the ARG after refinement. The eager strategy is used to implement eager predicate abstraction (AB) approaches, where the whole ARG is recomputed after new predicates are added to the domain.

SMT Solver Interface Components of the analysis phase use an SMT solver in a variety of ways:

- the ARG constructor uses it to check that the annotations of the ARG form a safe inductive invariant;
- abstract post, e.g., using predicate abstraction, may encode post computations as SMT queries; and
- the refiner can use it to find counterexamples and to compute interpolants.

All these uses are handled through a general interface to two SMT solvers: MathSAT5 [CGSS13] and Z3 [dMB08]. MathSAT5 is used for its native interpolation support. Z3 is favored for its efficiency and is used for all satisfiability queries (not requiring interpolation) and quantifier elimination for DAG interpolant computation.[1]

[1] At the time of implementation, Z3 did not have interpolant generation support.

7.3 Prominent Optimizations

We have incorporated in UFO_{app} a number of optimizations in order to improve performance of its various components. We describe the most prominent optimizations here:

Incremental Solving for Covering The ExpandArg algorithm, both in Vinta and Ufo, uses an SMT call to check if a node is covered. Since this is a heavily applied operation, a naive application of covering checks using independent calls to the SMT solver can be quite inefficient. In practice, we exploit Z3's incremental interface (using `push` and `pop` commands); it allows us to avoid performing SMT solving from scratch for similar queries.

For each cutpoint ℓ, we maintain a separate SMT context $\texttt{ctx}_\ell$. Every time a node v such that $\nu(v) = \ell$ is not covered (i.e., in Vinta, the check on line 53 in Algorithm 4 fails), $\neg\psi(v)$ is added to $\texttt{ctx}_\ell$. To check whether a node u with $\nu(u) = \ell$ is covered, we check whether $\psi(u)$ is satisfiable in $\texttt{ctx}_\ell$. If the result is unsatisfiable, then u is covered; otherwise, it is not covered and $\neg\psi(u)$ is added to $\texttt{ctx}_\ell$. Effectively, this is the same as checking whether

$$\psi(u) \wedge \bigwedge_{v \in \mathsf{vis}(\nu(u)), v \neq u} \neg\psi(v)$$

is unsatisfiable, which is equivalent to Vinta's line 53 of ExpandArg (Algorithm 4).

Using Post Computations for Simplification In our implementation, we keep track of those ARG edges for which $\textsc{Post}_{\mathcal{D}}$ computations returned $\bot$. For each such edge e, we can replace $\mathcal{L}_E(e)$ with *false*, where $\mathcal{L}_E(e)$ encodes semantics of an edge e in the ARG, as used in the refinement strategies of Ufo and Vinta (UfoRef 5 and VintaRef 6). We call such edges *false edges*, and exploit them to shrink encodings before passing them onto the SMT solver.

Improving Interpolation with UNSAT Cores One technical challenge we faced is that MathSat5's performance degrades significantly when interpolation support is turned on, particularly on large formulas encoding ARGs (see ARGCond in Section 3.3). To reduce the size of the formula given to MathSat5, we use the *assumptions* feature in the highly efficient Z3. Let a formula $\varphi_1 \wedge \ldots \wedge \varphi_n$ and a set $X = \{b_i\}_{i=1}^n$ of Boolean *assumption* variables be given. When Z3 is given a formula

$$\Phi = (b_1 \Rightarrow \varphi_1) \wedge \ldots \wedge (b_n \Rightarrow \varphi_n),$$

it returns a subset of X, called an *UNSAT core*, that has to be *true* to make Φ unsatisfiable. In our case, we add an assumption for each literal appearing in formulas in $\mathcal{L}_E$, and use Z3 to find unnecessary literals, i.e., those not in the UNSAT core. Since Z3 does not produce a minimal core, we heuristically repeat the minimization process three times. Finally, we set unnecessary literals to *true* and use MathSat5 to interpolate over the simplified formula. The simplified formula is weaker than the original formula, but by definition of UNSAT core is still unsatisfiable. In other words, the simplified formula over-approximates our encoding of program semantics. Therefore, a well-labeling of the ARG computed via the over-approximated semantics is still a well-labeling of the precise semantics.

7.4 Contributions in SV-COMP

In this section, we discuss UFO$_{app}$'s participation in the second and third editions of the International Competition on Software Verification (SV-COMP), a community initiative to benchmark and compare verification tools.

7.4.1 SV-COMP 2013

Benchmarks and Results The 2013 edition of SV-COMP [Bey13] contained a number of verification categories, which are sets of benchmarks on which verification tools are run and evaluated, primarily on the amount of correct results computed within an allotted time limit of 15 minutes of CPU time per benchmark. Each verification category contains programs with assertions that share common verification needs, for instance, requiring numerical invariants or concurrency support.

For SV-COMP 2013, we submitted an instantiation of UFO$_{app}$ that placed first in the four categories for which it was designed to work. The total number of categories was ten. UFO$_{app}$ received the gold medal in the following categories:

- `ControlFlowInteger`: a set of benchmarks including Microsoft Windows device drivers, SSH models, amongst others.
- `DeviceDrivers64`: a set of 64-bit Linux device drivers where the goal is to verify correct usage of APIs.
- `SystemC`: a set of benchmarks derived from sequentializing SystemC programs and translating them into C [CMNR10].
- `ProductLines`: a set of programs generated from *software product lines* [AvRW+13].

In the above categories, UFO$_{app}$ outperformed a number of prominent and highly-engineered tools including CPAchecker [BK11] and a new version of Blast [BHJM07]. In the `ControlFlowInteger` category, UFO$_{app}$ correctly solved *all* benchmarks in 450 seconds, whereas two different versions of CPAchecker took 1200 and 3400 seconds to solve benchmarks in this category. Both CPAchecker [Wen13, Löw13] tools did not solve the full set of benchmarks in `ControlFlowInteger`. We notice a similar trend in the other categories as well: UFO$_{app}$ is not only able to solve more benchmarks per category, but is highly competitive in terms of time taken per benchmark. A thorough presentation of the competition's results is available in Beyer's SV-COMP 2013 report [Bey13].

Categories in which UFO$_{app}$ did not participate include `Concurrency` and `MemorySafety`, which require reasoning about threads and heap shapes, two forms of analysis currently not supported by UFO$_{app}$.

Parallel UFO As we saw in Chapters 4 and 5, different instantiations of UFO$_{app}$ can produce drastically different performance. Given the diversity of the competitions benchmarks, there was no single instantiation of UFO$_{app}$'s abstract domains that was a clear winner. Therefore, for the purposes of the competition, we designed a parallel implementation of UFO$_{app}$ that runs seven instantiations of UFO$_{app}$ in parallel. These instantiations vary the abstract domain and the widening strategy, while always using Vinta's refinement strategy with restricted DAG interpolants.

Abstract domain	Widening	Time limit (s)
Boxes	3	20
Boxes	17	10
Cartesian Abs.	-	∞
Box	3	∞
Boolean Abs.	-	20
Box + Boolean Abs.	3	60

Table 7.1: Instantiations of UFO_{app} in SV-COMP 2013.

Since the competition's rules have a CPU time limit, we had to be careful not to incorporate ineffective instantiations. To that end, we devised a strategy that aborts certain instantiations after a set amount of time. Specifically, we found that on instantiations using very precise domains, like Boxes and Boolean predicate abstraction, if the analysis takes more than a minute, then it will likely not return a result within a reasonable amount of time. This is partly due to the fact that precise analyses quickly produce large ARGs whose encodings are very large formulas that SMT solvers cannot handle.

Table 7.1 shows the different instantiations of UFO_{app} used in parallel UFO_{app}.[2] Each row shows an instantiation as the abstract domain used, the widening strategy used in terms of when widening is applied (e.g., after three loop unrollings), and the time limit imposed on the instantiation. For instance, the first instantiation is the Boxes domain, where widening is applied every third iteration and the instantiation is aborted if does not return a result within 20 seconds. The last instantiation uses an abstract domain combining Box and Boolean abstraction, where the results of post are computed independently for each and conjoined after concretization.

7.4.2 SV-COMP 2014

In the 2014 edition of SV-COMP [Bey14], we submitted a similar version of UFO_{app} to that used in SV-COMP 2013. UFO_{app} won the silver medal in two of the verification categories, namely, `DeviceDrivers64` and `Simple`.

We attribute the relatively lower performance of UFO_{app} in SV-COMP 2014 to two primary factors:

- *Improvements in competing tools.* For instance, an efficient version of Blast, submitted by suppliers of the Linux device drivers benchmarks [ldv], won the `DeviceDrivers64` category, leaving UFO_{app} second.

- *New benchmarks that cannot be handled efficiently by our front-end.* For instance, new benchmarks were added to the `SystemC` category from SV-COMP 2013 that use arrays to model thread sequentialization. In theory, these arrays can be transformed into integer variables, but our front-end could not perform this transformation. This left our analysis algorithm unable to precisely model program semantics (as it does not currently have support for arrays). Additionally, new large Linux device drivers exposed a bottleneck in our front-end: a very long preprocessing phase and an unreasonably large program P^o.

[2] One of the configurations is elided as it just uses a slightly different preprocessing step to produce P^o.

7.5 Conclusion

In this chapter, we described UFO_{app}, a verification tool and framework that we implemented in the LLVM infrastructure for verifying C programs. UFO_{app} was used to implement and evaluate various instantiations of the Ufo and Vinta algorithms. UFO_{app}'s modular architecture makes it suitable for implementation and evaluation of different verification algorithms. For instance, by modifying the abstract domain or the refinement strategy, we can arrive at drastically different verification algorithms, each with different strengths and weaknesses.

Chapter 8

Conclusion

We have arrived at the end of this dissertation. In this chapter, we recap the highlights of this dissertation and sketch possible future paths to take from here.

8.1 Dissertation Summary

In this dissertation, we addressed the problem of automated software verification: proving that a program satisfies some property, for example, memory safety, termination, or functional correctness. This generally undecidable problem has received an enormous amount of attention from a wide array of computer science communities. Here, we focused on a promising class of algorithms called interpolation-based techniques, where Craig interpolants are used to hypothesize inductive invariants. Specifically, we advanced the state-of-the-art in several directions: (1) new intraprocedural interpolation-based verification techniques using the notion of DAG interpolants; (2) new interpolation-based techniques that utilize and complement results of classical abstract fixpoint invariant computation techniques; and (3) a new interprocedural verification technique that exploits Craig interpolants to compute procedure-modular proofs of correctness. The power of our conceptual and algorithmic contributions is demonstrated in UFO_{app}: an automated software verification tool for verifying (and finding bugs in) programs written in the C programming language. In its first participation in the International Competition on Software Verification (SV-COMP 2013), our UFO_{app} tool placed first in four out of ten verification categories, winning every benchmark category for which it was designed to work and outperforming state-of-the-art tools from the community. In the rest of this section, we recap the key technical contributions of this dissertation.

In Chapter 3, we introduced the notion of directed acyclic graph (DAG) interpolants and demonstrated how they can be utilized for intraprocedural verification. In interpolation-based software verification, interpolants are used to hypothesize an inductive invariant by proving correctness of finite (loop-free) paths through the control-flow graph of a program. DAG interpolants allow us to symbolically prove correctness of sets of program paths succinctly encoded as a DAG. Our main insight for computing DAG interpolants is a reduction of DAG interpolation to sequence interpolation. In other words, we linearize the DAG and treat it as if it is a sequence of instructions (a straight line program) and utilize previous techniques from the literature to compute a proof for the DAG-like program.

In Chapter 4, we addressed the problem of how to efficiently verify programs using DAG interpolants.

Specifically, we proposed a new algorithm, called Ufo, that iteratively unrolls the control-flow graph of a program into a DAG structure, called an abstract reachability graph (ARG), and uses DAG interpolants to label the ARG with a Hoare-style proof. We took Ufo further and incorporated abstraction-based techniques like predicate abstraction to unroll the program into an ARG and provide us with inductive invariants. The result was a hybrid algorithm that incorporates interpolation-based and abstraction-based verification techniques. Our experimental evaluation showed that hybrid instantiations of Ufo can outperform purely interpolation-based and abstraction-based techniques.

In Chapter 5, we proposed Vinta, an algorithm that extends Ufo in two directions. First, we extended Ufo to accept arbitrary abstract domains, as opposed to finite-height domains like predicate abstraction. This enabled greater flexibility in experimenting with a wide range of abstract domains. Second, we introduced the concept of restricted DAG interpolants, which enabled a tighter integration between results of interpolation and abstract interpretation. Whereas we set out with the goal of improving interpolation-based techniques, the Vinta algorithm can also be viewed as a technique for refining (strengthening) inductive invariants computed via abstract interpretation. Specifically, interpolants recover imprecision incurred by approximate operations employed by abstract domains, for example, join and widening.

In Chapter 6, we addressed the problem of verifying programs with procedures and recursion using interpolants, a question that had received little attention in the literature. Specifically, we presented Whale, a interprocedural verification algorithm that utilizes interpolants to compute proofs in the form of procedure summaries. Our key insight is that interpolants can be used to hypothesize a procedure summary by generalizing an under-approximation of a procedure (a finite unrolling). The properties of interpolants, particularly language restriction, provide us with concise summaries over a procedure's parameters and returns.

In Chapter 7, we presented and discussed the architecture, implementation, and optimizations of our UFO_{app} tool. UFO_{app} is a framework for building verification algorithms that is modelled after the Ufo and Vinta algorithms. UFO_{app} is parameterized mainly by the abstract domain used and the refinement strategy, providing us with flexible platform for experimenting with different instantiations of Ufo, Vinta, and others. Indeed, we utilized UFO_{app}'s flexibility to build a number of instantiations of Vinta using different domains that run in parallel to increase chances of proving safety or finding bugs. Our parallel version of UFO_{app} won the largest number of gold medals in SV-COMP 2013.

8.2 Future Outlook

Despite all of the advances made in this dissertation, a huge number of problems remain unanswered! In this section we discuss some of the most important and interesting directions for future research.

- **Supporting Concurrency** All techniques presented in this dissertation have been restricted to sequential programs—without concurrency. Of course, one could model interleavings between concurrent threads using an explicit scheduler, reducing the program to a sequential one. But such reduction results in highly complex sequential programs that are hard to analyze. Modular proof systems for concurrent programs were introduced to avoid explicitly modeling thread interleavings. For instance, Owicki-Gries [OG76] and Rely-Guarantee [Jon83] proofs can be computed using techniques similar to interprocedural analysis, as shown by Grebenshchikov et al. [GLPR12]. So far, though, techniques for computing modular proofs have been restricted to very simple programs

and properties. We believe the ideas presented in this dissertation can be adapted for concurrent program verification.

Using DAG interpolants, we showed how to examine a finite set of paths through the control-flow graph of a program and hypothesize a proof of the whole program. This raises the question of whether we can use a similar idea for concurrent program verification. Specifically, we need a succinct encoding of thread interleavings and a way to hypothesize proofs of correctness. Recent work by Sinha and Wang [SW11] presented an encoding of unrollings of two threads where interleavings are succinctly symbolically encoded. Sinha and Wang used the encoding for bounded verification. One question we could ask is whether we can extract a proof from bounded verification attempts. Specifically, given an unsatisfiable encoding of two unrolled threads (i.e., there are no bugs), can we use interpolants to hypothesize a proof of correctness? The first issue here is what proof system to use. In the case of sequential programs, we are typically interested in computing loop invariants. In a concurrent setting, there's a plethora of proof techniques (Owicki-Gries, Rely-Guarantee, iD-FGs [FKP13], etc.), and each one has its limitations. Second, sequential program encodings have a lot of structure: we can split the encoding into two halves, one encoding all executions before some statement in the program and another encoding all executions after it. This enables computing interpolants that describe states at a particular location. Succinct encodings of concurrent programs, like Sinha and Wang's [SW11], do not have such obvious structure. Given the power of SMT solvers and interpolation, it would be interesting to explore this direction for concurrent program verification.

- **Handling Deep Heap Properties** In this dissertation, we relied on first-order logic encodings of program semantics to compute first-order invariants. To prove memory safety and properties of heap-allocated data structures, first-order invariants often do not suffice. For instance, we might require an invariant that states that there is a linked list on the heap and all of its elements are greater than five. Verifying programs requiring such invariants is an interesting and important problem that has not received its fair share of research.

  In current work, we are exploring a combination of separation logic with interpolation. Separation logic allows us to describe shapes of heap-allocated data structures, like lists, trees, lists of lists, etc. Interpolants, on the other hand, provide us with descriptions of program data like integer variables. By combining the two, we are able to compute invariants that describe heap shape, program data, as well data stored in the heap. This line of work poses a large number of challenges: we need efficient decision procedures for rich fragments of separation logic with first-order constraints; we need ways of inferring heap shapes, preferably without fixing templates a priori as is commonly done; we need interpolation techniques that incorporate heap shape information described in separation logic; etc. We believe that this is a ripe and important area of research and that techniques in this dissertation can be profitably extended in this direction.

- **Verification in the Presence of Absence** In principle, an automated verification tool takes a program and automatically verifies it. This is not entirely true. Using automated verification tools often requires a significant manual effort to get the program into a shape acceptable by the tool. This is because programs typically use libraries and OS routines, some of which may be too complex to analyze or unavailable (if they are proprietary). Thus, stubs are often written to replace these unavailable pieces of the program. Stub writing is a time consuming and error prone

process, where a developer has to write a simple procedure that mimics behaviour of an original procedure and is sufficient for verification purposes. One interesting direction is to automate this process. Specifically, given a program and a property, can we synthesize the most permissive specification of unknown procedures that ensures the property holds? In other words, can we compute a conditional proof of program correctness: *the program satisfies the property assuming unknown procedures behave according to X*. Armed with such a technique, the job of the developer is now to simply check the inferred specifications/stubs, an arguably simpler process than coming up with stubs from scratch. We believe developing this research direction is essential for verification of large programs, which inevitably have complex dependencies.

- **Improving Interprocedural Verification** In Vinta, we showed how to utilize results of abstract interpretation to improve the interpolation process and vice versa. Our experience and evaluation of Ufo and Vinta indicated that the best configurations are those that use both abstract domains and interpolation. It is thus an interesting direction to explore this combination in an interprocedural setting, as in Whale. Specifically, it would be interesting to explore an approach analogous to Vinta's. First, an abstract domain is used to compute procedure summaries. Whether the analysis is performed top-down or bottom-up, the result will be an unrolling of the call graph of the program, an interprocedural ARG (iARG) that is annotated with procedure summaries. If procedure summaries are too weak to prove a property of interest, we could use state/transition or tree interpolants to strengthen the annotation and continue the fixpoint computation. Further, we could extend transition interpolants to *restricted transition interpolants*, in a manner analogous to RDI, where results of abstract interpretation are used to weaken the interpolation problem. It is our belief that such interprocedural analysis would improve upon purely interpolation-based techniques like Whale and Duality [MR13].

- **Tightening Interpolant–Abstract Domain Integration** In Vinta, we moved information between interpolants and the abstract domain. We assumed that the abstract domain is subsumed by the logic used for interpolation. For instance, all elements of the Box domain are representable in the theory of linear rational arithmetic (QF_LRA). On the other hand, converting (via abstraction function α) a formula in our logic to an abstract element may result in imprecision. For instance, QF_LRA contains disjunctions and non-linear constraints, which cannot be represented in Box. In our experiments, we implemented a simple and imprecise abstraction function. To avoid this imprecision, we need ways of computing *best* abstraction functions. In recent work [LAK+14], we addressed this problem for computing best abstractions of QF_LRA formulas in the family of template constraint matrix (TCM) domains, which include Box, octagons, etc. Specifically, we cast this problem as non-convex optimization and extended SMT solvers with optimization capabilities. In the future, it would be interesting to extend this technique to other theories like linear integer arithmetic (QF_LIA) and bitvector arithmetic (QF_BV). The appeal of this problem extends beyond its immediate application in our setting, as optimization is of importance in synthesis, bug finding, constraint programming, amongst others.

Bibliography

[AGC12a] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig Interpretation. In *Proceedings of the International Static Analysis Symposium (SAS)*, volume 7460 of *LNCS*, pages 300–316, 2012.

[AGC12b] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From Under-approximations to Over-approximations and Back. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *LNCS*, pages 157–173, 2012.

[AGC12c] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 672–678, 2012.

[AGC12d] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An Interpolation-based Algorithm for Inter-procedural Verification. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2012.

[AGL+13] Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. UFO: Verification with Interpolants and Abstract Interpretation - (Competition Contribution). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *LNCS*, pages 637–640, 2013.

[AKNR12] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. Parallelizing Top-down Interprocedural Analyses. In *Proceedings of theACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 217–228, New York, NY, USA, 2012.

[AvRW+13] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for Product-line Verification: Case Studies and Experiments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 482–491, 2013.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, June 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*. Springer, 1999.

[BCF+08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT Solver. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 299–303, 2008.

[BCG+09] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software Model Checking via Large-Block Encoding. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 25–32, 2009.

[Bey12] Dirk Beyer. Competition on Software Verification - (SV-COMP). In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 504–524, 2012.

[Bey13] Dirk Beyer. Second Competition on Software Verification - (Summary of SV-COMP 2013). In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 594–609, 2013.

[Bey14] Dirk Beyer. Status Report on Software Verification - (Competition Summary SV-COMP 2014). In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 373–388, 2014.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.

[BHZ06] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening Operators for Powerset Domains. *STTT*, 8(4-5):449–466, 2006.

[BK11] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 184–190, 2011.

[BNRS08] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs

from Tests. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 3–14, 2008.

[Bou93] François Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Proceedings of the International Conference on Formal Methods in Programming and Their Applications (FMPA)*, LNCS, pages 128–141, 1993.

[BR01] Tom Ball and Sriram Rajamani. The SLAM Toolkit. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 260–264, 2001.

[Bry86] Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *Transactions on Computers*, 8(C-35):677–691, 1986.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[CC76] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of the Colloque sur la Programmation*, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *LNCS*. Springer-Verlag, 1981.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the International Con-*

ference on Computer Aided Verification (CAV), volume 1855 of *LNCS*, pages 154–169, 2000.

[CGS10] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. *ACM Transactions on Computational Logic*, 12(1):7, 2010.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 93–107, 2013.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176, March 2004.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*, pages 570–574, 2005.

[cla] clang: a C language family frontend for LLVM. http://clang.llvm.org/. Accessed: July 27 2014.

[CMNR10] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Verifying SystemC: A Software Model Checking Approach. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 51–59, 2010.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[CPR05] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction Refinement for Termination. In *Proceedings of the International Static Analysis Symposium (SAS)*, volume 3672 of *LNCS*, pages 87–101, September 2005.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination Proofs for System Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426. ACM, 2006.

[Cra57] William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[Dij72] Edsger W. Dijkstra. Notes on Structured Programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 1–82, London, UK, 1972. Academic Press Ltd.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[DKPW10] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 129–145. Springer, 2010.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 337–340, 2008.

[EKS06] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 489–503, 2006.

[FKP13] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive Data Flow Graphs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 129–142, 2013.

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.

[GC10] Arie Gurfinkel and Sagar Chaki. Boxes: A Symbolic Abstract Domain of Boxes. In *Proceedings of the International Static Analysis Symposium (SAS)*, volume 6337 of *LNCS*, pages 287–303, 2010.

[GCNR08] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically Refining Abstract Interpretations. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 443–458, 2008.

[GCS11] Arie Gurfinkel, Sagar Chaki, and Samir Sapra. Efficient Predicate Abstraction of Program Summaries. In *Proceedings of the NASA Formal Methods Symposium (NFM)*, volume 6617 of *LNCS*, pages 131–145, 2011.

[GHK+06] Bhargav Gulavani, Thomas Henzinger, Yamini Kannan, Aditya Nori, and Sriram Rajamani. SYNERGY: A New Algorithm for Property Checking. In *Proceedings of ACM SIGSOFT*

Conference on Foundations of Software Engineering (FSE), pages 117–127, 2006.

[GLPR12] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing Software Verifiers from Proof Rules. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 405–416, 2012.

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 43–56, 2010.

[GPR11] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate Abstraction and Refinement for Verifying Multi-threaded Programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 331–344, 2011.

[Gri11] Alberto Griggio. Effective Word-level Interpolation for Software Verification. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 28–36, 2011.

[GS97] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 72–83, 1997.

[HHP10] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested Interpolants. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 471–482, 2010.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from Proofs. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 232–244. ACM, 2004.

[HJMS02] Thomas Henzinger, Ranjit Jhala, Rupdak Majumdar, and Gregoire Sutre. Lazy Abstraction. In *Proceedings of ACM SIGPLAN Symposium on the Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa71] C.A.R. Hoare. Procedures and Parameters: An Axiomatic Approach. In *Proceedings of Symposium on Semantics of Algorithmic Languages*, volume 188, pages 102–116, 1971.

[JM07] Ranjit Jhala and Kenneth McMillan. Array Abstractions from Proofs. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2007.

[Jon83] Cliff B. Jones. Specification and Design of (Parallel) Programs. In *Proceedings of the IFIP World Computer Congress*, pages 321–332, 1983.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KW07] Daniel Kroening and Georg Weissenbacher. Lifting Propositional Interpolants to the Word-Level. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 85–89. IEEE Computer Society, 2007.

[KW11] Daniel Kroening and Georg Weissenbacher. Interpolation-Based Software Verification with Wolverine. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 573–578, 2011.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–88, 2004.

[LAK+14] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic Optimization with SMT Solvers. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 607–618, 2014.

[ldv] Linux Driver Verification. http://linuxtesting.org/project/ldv. Accessed: July 27 2014.

[Löw13] Stefan Löwe. CPAchecker with Explicit-Value Analysis Based on CEGAR and Interpolation - (Competition Contribution). In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 610–612, 2013.

[LW12] Stefan Löwe and Philipp Wendler. CPAchecker with Adjustable Predicate Analysis - (Competition Contribution). In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 528–530, 2012.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[McM03] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 1–13, 2003.

[McM04] Kenneth L. McMillan. An Interpolating Theorem Prover. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988, pages 16–30. Springer, 2004.

[McM06] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 123–136, 2006.

[McM10] Kenneth McMillan. Lazy Annotation for Program Testing and Verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 104–118, 2010.

[Min06] Antoine Miné. The Octagon Abstract Domain. *Journal of Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[MM70] Zohar Manna and John McCarthy. Properties of Programs and Partial Function Logic. *Journal of Machine Intelligence*, 5, 1970.

[MR13] Kenneth L. McMillan and Andrey Rybalchenko. Computing Relational Fixed Points using Interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.

[MZ09] Sharad Malik and Lintao Zhang. Boolean Satisfiability From Theoretical Hardness to practical Success. *Communications of the ACM*, 52(8):76–82, 2009.

[NMRW02] George Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analsysis and Transformation of C Programs. In *Proceedings of the International Conference on Compiler Construction (CC)*, volume 2304 of *LNCS*, pages 213–228. Springer, April 2002.

[OG76] Susan S. Owicki and David Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, 6:319–340, 1976.

[pac] Pacemaker Formal Methods Challenge. http://sqrl.mcmaster.ca/pacemaker.htm. Accessed: July 27 2014.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the Annual Symposium on the Foundations of Computer Science*, pages 46–57, 1977.

[QS81] Jean-Pierre Quielle and Joseph Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the International Symposium in Programming*, pages 337–351, 1981.

[Rep96] Thomas W. Reps. On the Sequential Nature of Interprocedural Program-Analysis Problems. *Acta Informatica*, 33(8):739–757, 1996.

[RHK13a] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Classifying and Solving Horn Clauses for Verification. In *Proceedings of the International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *LNCS*, pages 1–21. Springer, 2013.

[RHK13b] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive Interpolants for Horn-Clause Verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.

[RHS95] Thomas W. Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages(POPL)*, pages 49–61, 1995.

[SP81] Micha Sharir and Amir Pnueli. *Program Flow Analysis: Theory and Applications*, chapter Two Approaches to Interprocedural Data Flow Analysis, pages 189–233. Prentice-Hall, 1981.

[SSM05] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable Analysis of Linear Systems using Mathematical Programming. In *Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 25–41, 2005.

[SW11] Nishant Sinha and Chao Wang. On Interference Abstractions. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 423–434, 2011.

[Tur36] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Tur49] Alan M. Turing. Checking a Large Routine. In *Report on a Conference on High Speed Automatic Computation, June 1949*, pages 67–69, Cambridge, UK, 1949. University Mathematical Laboratory, Cambridge University. Inaugural conference of the EDSAC computer at the Mathematical Laboratory, Cambridge, UK.

[WCC+12] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined Behavior: What Happened to My Code? In *Proceedings of the Asia-Pacific Workshop on Systems (APSYS)*, pages 9:1–9:7. ACM, 2012.

[Wei12] Georg Weissenbacher. Interpolant Strength Revisited. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of

LNCS, pages 312–326. Springer, 2012.

[Wen13] Philipp Wendler. CPAchecker with Sequential Combination of Explicit-State Analysis and Predicate Analysis - (Competition Contribution). In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 613–615, 2013.

[Won12] Daniel Wonisch. Block Abstraction Memoization for CPAchecker - (Competition Contribution). In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 531–533, 2012.

[WYGI07] Chao Wang, Zijiang Yang, Aarti Gupta, and Franjo Ivancic. Using Counterexamples for Improving the Precision of Reachability Computation with Polyhedra. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 352–365, 2007.

[WZKSL13] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 260–275, 2013.

[YFCW14] Ming-Hsien Tsai Bow-Yaw Wang Yu-Fang Chen, Chiao Hsieh and Farn Wang. Verifying Recursive Programs using Intraprocedural Analyzers, 2014.

Appendices

Appendix A

Inductive Invariants Example

We provide an informal description of safe inductive invariants as a mechanism for proving programs correct, and briefly discuss how a software verifier can find such invariants.

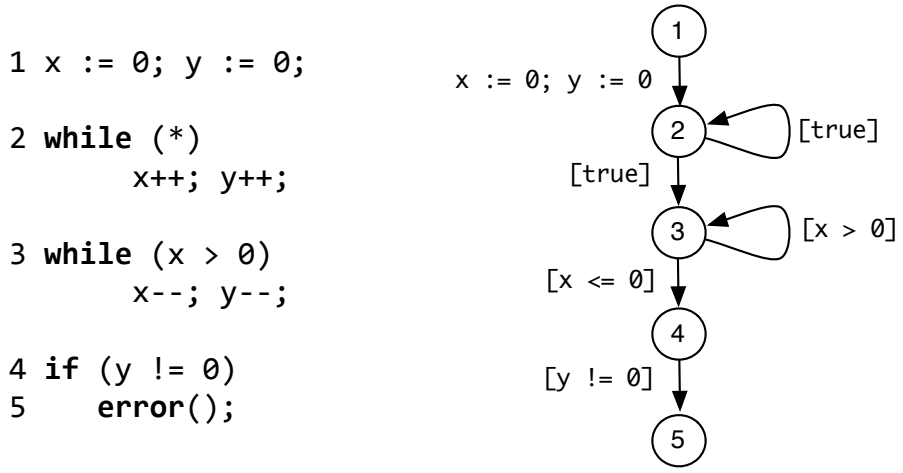
Consider the program in Figure A.1(a), where `*` denotes a nondeterministic choice. Our goal is to prove that the call to `error()` is unreachable. To do so, we require a safe inductive invariant. An inductive invariant is an annotation of each location (line) in the program describing an over-approximation of the set of reachable states at that location, where a state is a valuation of all program variables. Additionally, annotations of two consecutive program locations must form a valid *Hoare triple* [Hoa69]. For example, an inductive invariant might specify that $x \geqslant 0$ at location 2, since when an execution first arrives at location 2 the value of `x` is `0`, and then after each loop iteration the value of `x` just increases. The annotation of location 2 forms a valid Hoare triple with itself, denoted

$$\{x \geqslant 0\}\ \mathtt{x++;y++}\ \{x \geqslant 0\}.$$

This Hoare triple specifies that if we start in any state such that `x` is greater than or equal to 0 (as specified on the left side of the triple) and we execute $\mathtt{x++;y++}$, we will definitely arrive at a state where `x` is greater than or equal to 0 (as specified on the right side of the triple). Note that we are assuming that the variables are of type $\mathbb{Z}$, therefore, no overflows or underflows occur. A safe inductive invariant is one where the error location, location 5 in our case, is labeled by *false*, implying that no execution can reach that location.

Consider the annotation **Inv1** of our program shown in Table A.1. The annotation represents an inductive invariant. Note that location 5 is labeled by $x \geqslant 0$, implying that executions may reach the error location. Thus, **Inv1** does not preclude unsafe states and is an unsafe inductive invariant. **Inv2** is another inductive invariant for the program in Figure A.1 that labels location 5 with *false*, proving that no execution can reach `error()`. Thus, **Inv2** is a safe inductive invariant.

Given our example program, an automated verifier based on predicate abstraction typically starts by constructing an invariant using a restricted language of predicates, e.g., invariants only including the predicate $x \geqslant 0$. Using the predicate $x \geqslant 0$, the strongest logical formula that can be used to annotate location 4 is $x \geqslant 0$ (as in **Inv1**). This is a very coarse over-approximation of reachable states at location 4, since a state satisfying $x \geqslant 0$ may also satisfy $y \neq 0$ (the guard of the if-statement); therefore, the verification tool cannot annotate location 5 with *false*. When a verifier employing a CEGAR loop [CGJ+00] constructs an unsafe invariant, it checks if the program is indeed unsafe, i.e.,



(a) (b)

Figure A.1: Example program and its control-flow-graph representation.

Location	**Inv1**	**Inv2**
1	*true*	*true*
2	$x \geqslant 0$	$x = y \wedge x \geqslant 0$
3	$x \geqslant 0$	$x = y \wedge x \geqslant 0$
4	$x \geqslant 0$	$x = y \wedge x = 0$
5	$x \geqslant 0$	$false$

Table A.1: Safe and unsafe inductive invariants.

that there is an erroneous execution. If it cannot find such an execution, it adds more predicates to the language of the invariant and constructs a new invariant. In this example, the verifier might discover the predicates $x = 0$ and $x = y$, enabling it to compute **Inv2** and prove the program safe.

Appendix B

Proofs

In this appendix, we prove lemmas and theorems whose proofs do not appear, or are sketched, in the main text.

B.1 Proof of Theorem 2.1

Theorem (Program Safety)**.** *If there exists a safe, complete, and well-labeled ARG for a program P, then P is safe.*

Proof. Suppose we are given a safe, complete, well-labeled ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$ of a program $P = (\mathcal{L}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$. Then, we want to show that

$$Inv = \{\ell \mapsto I_\ell \mid \ell \in \mathcal{L}\}, \text{where}$$

$$I_\ell = \bigvee\{\psi(v) \mid v \in V \text{ and } \nu(v) = \ell \text{ and } v \text{ is uncovered}\},$$

is a safe inductive invariant of P.

By definition of a safe and well-labeled ARG, we know that

1. $Inv(\mathsf{en}) = \mathit{true}$, since $\psi(v_{\mathsf{en}}) = \mathit{true}$; and
2. $Inv(\mathsf{err}) = \mathit{false}$, since $\psi(v) = \mathit{false}$, for all $v \in V$ such that $\nu(v) = \mathsf{err}$.

We now need to show that for any $(\ell_1, T, \ell_2) \in \delta$,

$$Inv(\ell_1) \wedge [\![T]\!] \Rightarrow Inv(\ell_2)'.$$

By our construction of Inv, we know that

$$Inv(\ell_1) = \bigvee\{\psi(v) \mid v \in V \text{ and } \nu(v) = \ell_1 \text{ and } v \text{ is uncovered}\}$$

$$Inv(\ell_2) = \bigvee\{\psi(v) \mid v \in V \text{ and } \nu(v) = \ell_2 \text{ and } v \text{ is uncovered}\}$$

Let $u \in V$ be an uncovered node such that $\nu(v) = \ell_1$. By completeness of $\mathcal{A}$, this means that there

exists a vertex $v \in V$ such that $\nu(v) = \ell_2$. By well-labeledness of $\mathcal{A}$, we know that

$$\psi(u) \wedge [\![T]\!] \Rightarrow \psi(v)'.$$

By definition of $Inv(\ell_1)$, $\psi(u)$ is one of the disjuncts in $Inv(\ell_1)$. If v is uncovered, then $\psi(v)$ is a disjunct in $Inv(v)$. If v is covered, then we have to consider two cases. Recall definition of covered: there exists a node $q \in V$ that dominates u and there exists a set of nodes $X \subseteq V$ such that

- $\forall x \in X \cdot x$ is uncovered,
- $\psi(q) \Rightarrow \bigvee_{x \in X} \psi(x)$, and
- $\forall x \in X \cdot \nu(q) = \nu(x) \wedge q \not\preceq x$,

We consider the following two cases:

1. If $q \neq v$, then q must also dominate u, but u is uncovered, so it cannot be the case.
2. If $q = v$, then there is a set X of uncovered nodes such that (1) $\psi(q) \Rightarrow \bigvee_{x \in X} \psi(x)$ and (2) for all $x \in X$, $\nu(x) = \nu(v)$ (as per conditions of covering above). Since nodes in X are uncovered,

   $$\bigvee_{x \in X} \psi(x) \Rightarrow Inv(\ell_2).$$

   This means that

   $$\psi(u) \wedge [\![T]\!] \Rightarrow Inv(\ell_2)'.$$

Therefore, for any $(\ell_1, T, \ell_2) \in \delta$,

$$Inv(\ell_1) \wedge [\![T]\!] \Rightarrow Inv(\ell_2)'.$$

We have shown that a safe, complete, well-labeled ARG constitutes a safe inductive invariant, and therefore proves that P is safe. □

B.2 Proof of Theorem 5.1

Theorem (WidenWith$_{\{\vee,\sqcup\}}$ Correctness)**.** WidenWith$_{\sqcup}$ *and* WidenWith$_{\vee}$ *satisfy the two conditions of Definition 5.2.*

Proof. We prove correctness of WidenWith$_{\sqcup}$ and WidenWith$_{\vee}$ separately. Our proofs rely on the standard definition of the widening operator $\triangledown$ (where $\sqsubseteq$ is the abstract order):

- For all $x, y \in D$, $x \sqsubseteq x \triangledown y$ and $y \sqsubseteq x \triangledown y$.
- For any sequence $y_0, \ldots$, where $y_i \in D$, the sequence $x_0, \ldots$, where $x_0 = \bot$ and $x_i = x_{i-1} \triangledown y_{i-1}$ converges, i.e., $\exists i \cdot x_i \sqsubseteq x_{i-1}$.

WidenWith$_{\sqcup}$ Proof

First we prove soundness and then termination, as per Definition 5.2.

Soundness: We need to show that for any $X \subseteq D$ and $y \in D$,

$$(\gamma(X) \vee \gamma(y)) \Rightarrow (\gamma(X) \vee \gamma(X \triangledown_W y)).$$

It suffices to show that

$$\gamma(y) \Rightarrow \gamma(X \triangledown_W y). \tag{B.1}$$

In the case that $X = \emptyset$, we know that $(X \triangledown_W y) = y$, as per definition of WidenWith$_{\sqcup}$. Therefore, Formula B.1 is valid.

In the case that $X \neq \emptyset$, we know that $X \triangledown_W y = x \triangledown (x \sqcup y)$, where x is an arbitrary element of X. By definition of $\triangledown$, $x \sqsubseteq (X \triangledown_W y)$ and $y \sqsubseteq (X \triangledown_W y)$. Therefore, Formula B.1 is valid.

Termination: We want to show that for any $X \subseteq D$ and sequence $\{y_i\}_i \in D$, the sequence $\{Z_i\}_i \subseteq D$, where $Z_0 = X$ and $Z_i = Z_{i-1} \cup \{Z_{i-1} \triangledown_W y_i\}$, converges.

Fix some i. Now, consider any $z \in Z_i$. Either $z \in X$ or $\exists z' \in Z_{i-1} \cdot z = z' \triangledown y$, by definition of WidenWith$_{\sqcup}$. By definition of $\triangledown$ and by the fact the X is a finite set, there must be a $j > i$ such that for every $z_j \in Z_j$, there exists $z_i \in Z_i$ where $\gamma(z_j) \Rightarrow \gamma(z_i)$. Therefore, there must be a $j > i$ such that $\gamma(Z_j) \Rightarrow \gamma(Z_i)$.

WidenWith$_{\vee}$ Proof

Soundness: We need to show that for any $X \subseteq D$ and $y \in D$,

$$(\gamma(X) \vee \gamma(y)) \Rightarrow (\gamma(X) \vee \gamma(X \triangledown_W y)).$$

In the case that $X = \emptyset$, it is the same as the WidenWith$_{\sqcup}$ case.

In the case that $X \neq \emptyset$, we know that

$$X \triangledown_W y = \left((\bigvee X) \triangledown (\bigvee X \vee y)\right) \setminus \bigvee X,$$

as per definition of WidenWith$_{\vee}$. By definition of $\triangledown$, we know that

$$\gamma(X \triangledown_W y) \Leftarrow \gamma\left(y \setminus \bigvee X\right). \tag{B.2}$$

Since we are dealing with a disjunctive domain, we know that

$$\gamma\left(y \setminus \bigvee X\right) \equiv \gamma(y) \wedge \neg\gamma\left(\bigvee X\right). \tag{B.3}$$

From Formulas B.2 and B.3, it follows that soundness holds for WidenWith$_{\vee}$.

Termination: We want to show that for any $X \subseteq D$ and sequence $\{y_i\}_i \in D$, the sequence $\{Z_i\}_i \subseteq D$, where $Z_0 = X$ and $Z_i = Z_{i-1} \cup \{Z_{i-1} \triangledown_W y_i\}$, converges.

By definition of WidenWith$_\vee$, we know that

$$Z_i = Z_{i-1} \cup \left\{ \left(\bigvee Z_{i-1} \triangledown \left(\bigvee Z_{i-1} \vee y_i \right) \right) \setminus \bigvee Z_{i-1} \right\}$$

Since the set Z_i is removed (using $\setminus$) and added back (using $\cup$), we can simplify the above to

$$Z_i = \left(\bigvee Z_{i-1} \right) \triangledown \left(\bigvee Z_{i-1} \vee y_i \right).$$

By definition of $\triangledown$, this sequence converges. □

B.3 Proof of Theorem 5.2

Theorem (Correctness of VintaRef)**.** VintaRef *satisfies the specification of* Refine *in Definition 5.1.*

Proof. Our proof assumes soundness and completeness of the oracle ComputeDItp, which computes DAG interpolants. That is, ComputeDItp returns DAG interpolants (a nonempty set DItp) if and only if for every path $v_1, \ldots, v_n$ (where $v_1 = v^{en}$ and $v_n = v^{ex}$),

$$\left(\bigwedge_{i \in [1, n-1]} \mathcal{L}_E(v_i, v_{i+1}) \right) \Rightarrow \mathit{false}.$$

Correctness of ComputeRDItp

First, we prove that ComputeRDItp (Algorithm 6) indeed computes restricted DAG interpolants for a graph G with variable and edge labelings $\mathcal{L}_V$ and $\mathcal{L}_E$.

The first step of ComputeRDItp is to update the map $\mathcal{L}_E$ to $\mathcal{L}'_E$ in the loop at line 11. For each $e = (u, v) \in E$,

$$\mathcal{L}'_E(e) = \mathcal{L}_V(u) \wedge \mathcal{L}_E(e).$$

Then DAG interpolants are computed for G and $\mathcal{L}'_E$ and stored in RDItp.

Suppose RDItp $\neq \emptyset$. Then, by Definition 3.1 of DAG interpolants, RDItp$(v^{en}) = \mathit{true}$ and RDItp$(v^{ex}) = \mathit{false}$. This satisfies conditions 2 and 3 of Definition 5.3. Similarly, by definition of DAG interpolants, we know that for each $(u, v) \in E$,

$$\text{RDItp}(u) \wedge \mathcal{L}'_E(u, v) \Rightarrow \text{RDItp}(v).$$

By construction of $\mathcal{L}'_E$, this means that

$$\text{RDItp}(u) \wedge \mathcal{L}_V(v) \wedge \mathcal{L}_E(u, v) \Rightarrow \text{RDItp}(v). \tag{B.4}$$

By precondition of ComputeRDItp, $\mathcal{L}_V$ is a well-labeling of G. Therefore, for each $(u, v) \in E$,

$$\mathcal{L}_V(u) \wedge \mathcal{L}_E(u, v) \Rightarrow \mathcal{L}_V(v). \tag{B.5}$$

It follows from Formulas B.4 and B.5 that for all $(u, v) \in E$,

$$(\text{RDItp}(u) \wedge \mathcal{L}_V(v) \wedge \mathcal{L}_E(u, v)) \Rightarrow (\mathcal{L}_V(v) \wedge \text{RDItp}(v)).$$

This satisfies condition 1 of Definition 5.3.

The only thing that is left to show is condition 4 of Definition 5.3. This holds when the vertex labeling $\mathcal{L}_V$ satisfies the condition:

$$\forall v_i \in V \cdot FV(\mathcal{L}_V(v_i)) \subseteq \left(\bigcup_{e \in desc(v_i)} FV(\mathcal{L}_E(e)) \right) \cap \left(\bigcup_{e \in anc(v_i)} FV(\mathcal{L}_E(e)) \right)$$

Since the precondition of ComputeRDItp does not enforce this on condition on $\mathcal{L}_V$, it is not guaranteed that resulting restricted DAG interpolants satisfy condition 4. Nonetheless, correctness of VintaRef is not affected by this. □

Correctness of VintaRef

Assuming ComputeRDItp did not return an empty set, then VintaRef satisfies the specification of Refine (Definition 5.1). This follows from the definition of restricted DAG interpolants (Definition 5.3), the correctness of the function ComputeRDItp, and the correctness of the function DecodeBmc.

Now, suppose that ComputeRDItp returned an empty map. This means that there is a path $v_1, \ldots, v_n$ in G, from the entry node to the exit node, such that

$$\bigwedge_{i \in [1,n]} \mathcal{L}_V(v_i) \wedge \mathcal{L}_E(v_i, v_{i+1})$$

is satisfiable. It follows that

$$\bigwedge_{i \in [1,n]} \mathcal{L}_E(v_i, v_{i+1})$$

is satisfiable. By correctness of the BMC encoding of program semantics (EncodeBmc), it follows that there is a feasible execution to the error location.

B.4 Proof of Lemma 6.1

Lemma. *Given an iARG $\mathcal{I}^{\mathcal{A}}(P)$, an ARG $\mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P)$, and a set of exit nodes X, there exists a total onto map from satisfying assignments of* iARGCond$(\mathcal{A}_i, X)$ *to interprocedural (ϵ_i, X)-executions in $\mathcal{I}^{\mathcal{A}}(P)$.*

Proof. Let $\mathcal{I}^{\mathcal{A}}(P)$, an iARG of some program P, be of arbitrary size, $\mathcal{A}_i$ be some ARG in $\mathcal{I}^{\mathcal{A}}(P)$, and X be a set of exit nodes. We prove this lemma by induction on the depth of satisfying assignments – or the depth recursion in iARGCond$(\mathcal{A}_i, X)$. Depth n recursion means that for recursive calls at depth n, iARGCond$(\mathcal{A}_i, X)$ is replaced by ARGCond$(\mathcal{A}_i, X)$, i.e., call edges are unconstrained (non-deterministic).

Base Case: For depth 0,

$$\Phi_0 = \text{iARGCond}(\mathcal{A}_i, X) = \text{ARGCond}(\mathcal{A}_i, X) = C \wedge D.$$

Let Z be some satisfying assignment for Φ_0. Then, by definition of constraints C, there exists a sequence of Booleans $c_{v_1}, \ldots, c_{v_l}$ that are set to *true* in Z, where $v_1 = \epsilon_i$, v_l is an exit node in X, and there is a path $v_1, \ldots, v_l$ in $\mathcal{A}_i$.

By definition of D, for each edge represented by (c_{v_a}, c_{v_b}) along $c_{v_1}, \ldots, c_{v_l}$, the corresponding formula $[\![\tau(v_a, v_b)]\!]$ has to be satisfiable by Z. Therefore, by our SSA assumption, there exists an execution along $\nu(v_1), \ldots, \nu(v_l)$ where call statements are treated as non-deterministic assignments. This proves that there is a total map.

To prove that the map is onto, suppose there is a feasible execution of P starting in $\nu(\epsilon_i)$ and ending in a location corresponding to an exit node $v \in X$, where every call statement is treated non-deterministically. Let $v_1, \ldots, v_l$ be the path traversed by the execution. To make C satisfiable, set $c_{v_1}, \ldots, c_{v_l}$ to *true* and all other variables c_v to $false$. Under these constraints, to make D satisfiable, for each edge variable assigned along the execution, set its corresponding variable in the formula to the value it holds in the execution. All unassigned variables can hold any value. This satisfies every formula $[\![\tau(v_a, v_b)]\!]$, where $a = b - 1$ and $1 < b \leqslant l$. Therefore, D holds. Note that due to our SSA assumption, each value is assigned once.

Inductive Hypothesis: Assume Lemma holds for depth n of recursion.

Inductive Step: For depth $n + 1$,

$$\Phi_{n+1} = \textsc{iargCond}(\mathcal{A}_i, X) = C \wedge D \wedge \bigwedge_{j=1}^{m} \mu_j.$$

Let Z be a depth $n + 1$ assignment. That is, Z represents a path through $\mathcal{I}^{\mathcal{A}}(P)$ of depth $n + 1$. Therefore, Z is also a satisfying assignment for Φ_n, since in Φ_n all call statements at level $n + 1$ are treated non-deterministically. The only difference between Φ_{n+1} and Φ_n are the additional constraints for call statements of depth $n + 1$. By the inductive hypothesis and base case, there exists an execution of depth $n + 1$ through $\mathcal{I}^{\mathcal{A}}(P)$ corresponding to Z.

Suppose there is an execution of depth $n + 1$ through $\mathcal{I}^{\mathcal{A}}(P)$. By the inductive hypothesis, there is a satisfying assignment Z for Φ_n. To extend Z to a satisfying assignment Z' for Φ_{n+1}, set the variables appearing in the constraints of calls at level $n + 1$ to their corresponding values from the execution. By the base case, Z' satisfies Φ_{n+1}. □

B.5 Proof of Lemma 6.2

Lemma. *Given an iARG $\mathcal{I}^{\mathcal{A}}(P)$, an ARG $\mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P)$, a set of exit nodes X, and a sequence of formulas $I = \{(q_k, t_k)\}_{k=1}^{m}$, there exists a total and onto map from satisfying assignments of* $\textsc{SpecCond}(\mathcal{A}_i, X, I)$ *to (ϵ_i, X)-executions in $\mathcal{A}_i$, where each call-edge e_k is interpreted as* $\texttt{assume}(q_k \Rightarrow t_k)$.

Proof. This holds trivially from the proof of Lemma 6.1, since SpecCond is the same as a depth 1 iARGCond, where bodies of callees are replaced by an assume statement. □

B.6 Proof of Lemma 6.3

Lemma. *Given an ARG $\mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P)$, and a set of exit nodes X, let $\Phi = G_i \wedge \text{IARGCOND}(\mathcal{A}_i, X) \wedge \neg S_i$ be unsatisfiable and let $g_0, s_0, \ldots, s_m, g_{m+1}$ be $\text{STITP}(\Phi)$. Then,*

$$G_i \wedge \text{SPECCOND}(\mathcal{A}_i, X, \{(\text{GUARD}(g_k), \text{SUM}(s_k))\}_{k=1}^{m}) \wedge \neg S_i$$

is unsatisfiable.

Proof. Given some ARG $\mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P)$, let

$$\Phi = G_i \wedge \text{IARGCOND} \wedge \neg S_i = G_i \wedge \varphi \wedge \mu_1 \wedge \cdots \wedge \mu_m \wedge \neg S_i.$$

Suppose Φ is UNSAT and $\text{STITP}(\Phi) = g_0, s_0, \ldots, s_m, g_{m+1}$. We prove that

$$G_i \wedge \text{SPECCOND}\left(\mathcal{A}_i, X, \{(\text{GUARD}(g_k), \text{SUM}(s_k)\}_{j=1}^{m}\right) \wedge \neg S_i \text{ is UNSAT}$$

by transforming Φ into it, ensuring unsatisfiability at every step.

By the definition of state/transition interpolants,

$$G_i \wedge \varphi \wedge s_1 \wedge \cdots \wedge s_m \wedge \neg S_i \text{ is UNSAT.} \tag{B.6}$$

Note that s_k is over the variables $\vec{a}_k, \vec{b}_k$ and c_{v_k}, c_{w_k}. (See Chapter 6 for a description of these variables.) Therefore, we can replace occurrences of $\vec{a}_k, \vec{b}_k$ with $\vec{p}_k, \vec{r}_k$ in each s_k. and add the constraint $\chi_k = (\vec{a}_k = \vec{p}_k \wedge \vec{r}_k = \vec{b}_k)$ to each conjunct s.t.

$$G_i \wedge \varphi \wedge (\chi_1 \wedge s'_1) \wedge \cdots \wedge (\chi_m \wedge s'_m) \wedge \neg S_i \text{ is UNSAT,} \tag{B.7}$$

where $s'_k = s_k[\vec{a}_k, \vec{b}_k \leftarrow \vec{p}_k, \vec{r}_k]$.

Now, for each conjunct $(\chi_k \wedge s'_k)$, we transform it into $(c_{v_k} \wedge c_{w_k}) \Rightarrow (\chi_k \wedge s''_k)$, where $s''_k = s'_k[c_u \leftarrow \mathit{true} \mid c_u \text{ is a control variable}]$:

$$G_i \wedge \varphi \wedge ((c_{v_1} \wedge c_{w_1}) \Rightarrow (\chi_1 \wedge s''_1)) \wedge \cdots \wedge ((c_{v_m} \wedge c_{w_m}) \Rightarrow (\chi_m \wedge s''_m)) \wedge \neg S_i \text{ is UNSAT} \tag{B.8}$$

We show that (3) is UNSAT by showing that for an interpolant s_k, $s_k \equiv (c_{v_k} \wedge c_{w_k}) \Rightarrow s_k[c_{v_k}, c_{w_k} \leftarrow \mathit{true}]$. By definition of s_k,

$$\neg c_{v_k} \Rightarrow s_k \text{ and } \neg c_{w_k} \Rightarrow s_k. \tag{B.9}$$

We show the equivalence as follows:

	s_k	case splitting
$\Leftrightarrow$	$(\neg c_{v_k} \wedge s) \vee (\neg c_{w_k} \wedge s) \vee (c_{v_k} \wedge c_{w_k} \wedge s_k)$	using (B.9)
$\Leftrightarrow$	$\neg c_{v_k} \vee \neg c_{w_k} \vee (c_{w_k} \wedge c_{v_k} \wedge s_k)$	material implication
$\Longleftrightarrow$	$(c_{v_k} \wedge c_{w_k}) \Rightarrow c_{v_k} \wedge c_{w_k} \wedge s_k$	substitution
$\Leftrightarrow$	$(c_{v_k} \wedge c_{w_k}) \Rightarrow s_k[c_{v_k}, c_{w_k} \leftarrow \mathit{true}]$	

Note that $\varphi \equiv \varphi_1 \vee \cdots \vee \varphi_l$, where each φ_j represents a single path through $\mathcal{A}_i$ to X. From the definition of state interpolants, it follows that for every φ_j, $\rho_j^k \Rightarrow g_{k+1}$, where

$$\rho_j^k = G_i \wedge \varphi_j \wedge ((c_{v_1} \wedge c_{w_1}) \Rightarrow (\chi_1 \wedge s_1'')) \wedge \ldots ((c_{v_k} \wedge c_{w_k}) \Rightarrow (\chi_k \wedge s_k''))$$

Our goal now is to show that $\rho_j^k \Rightarrow \textsc{Guard}(g_{k+1})$. We start by showing that for all j, k,

$$\rho_j^k \Rightarrow g_{k+1}[c_u \leftarrow \mathit{true} \mid v_{k+1} \sqsubseteq u].$$

First, assume that φ_j represents a path that does not go through the $k+1$st call edge. If node u is reachable from v_{k+1} and is on path j, then any satisfying assignment to ρ_j^k forces c_u to be *true*. So $\rho_j^k \Rightarrow g_{k+1}[c_u \leftarrow \mathit{true} \mid v_{k+1} \sqsubseteq u]$. That is, we set all nodes on the path to true, as well as all other nodes reachable from v_{k+1}. Note that for every node u reachable from v_{k+1} and not appearing on the path j, its control variabl c_u does not appear in ρ_j^k. Therefore, the transformation preserves our desired implication.

Now suppose that node u is reachable from v_{k+1}, but not on path j, then c_u is not a variable in ρ_j^k (by property of interpolants and topological ordering of call formulae.) So $\rho_j^k \Rightarrow g_{k+1}[c_u \leftarrow \mathit{true} \mid v_{k+1} \sqsubseteq u$ and u is not on path $j]$.

Second, assume j is a path that does not go through the $k+1$st edge. If u is reachable from v_{k+1}, then u is on the current path or c_u is not in ρ_k^j. Therefore, $\rho_j^k \Rightarrow g_{k+1}[c_u \leftarrow \mathit{true} \mid v_{k+1} \sqsubseteq u]$.

From this, we know that for all j, k,

$$\rho_j^k \Rightarrow g_{k+1}[c_u \leftarrow \mathit{true} \mid v_{k+1} \sqsubseteq u].$$

By the above, we know that

$$\Phi_j = G_i \wedge \varphi_j \wedge ((c_{v_1} \wedge c_{w_1}) \Rightarrow (\chi_1 \wedge (g_1' \Rightarrow s_1'')) \wedge \cdots \wedge ((c_{v_m} \wedge c_{w_m}) \Rightarrow (\chi_m \wedge (g_m' \Rightarrow s_m'')) \wedge \neg S_i \text{ is UNSAT}, \quad \text{(B.10)}$$

where $g_k' = \exists Q \cdot g_{k+1}[c_u \leftarrow \mathit{true} \mid v_{k+1} \sqsubseteq u][\vec{b}_{k+1} \leftarrow \vec{p}_{k+1}]$. Q here refers to all variables in g_{k+1} except node Booleans (of form c_v) and $\vec{p}_{k+1}$. Existential quantification relaxes the formula and does not affect unsatisfiability. It follows from (4) that $\Phi_j[c_u \leftarrow \mathit{false} \mid u$ not on path $j]$ is also UNSAT. By definition of Guard,

$$\Phi_j[g_k' \leftarrow \textsc{Guard}(g_k) \mid \text{ for all } k] \Rightarrow \Phi_j[c_u \leftarrow \mathit{false} \mid u \text{ not on path } j].$$

□

By definition of SpecCond, it follows that:

$$G_i \wedge \textsc{SpecCond}\left(\mathcal{A}_i, X, \{(\textsc{Guard}(g_k), \textsc{Sum}(s_k)\}_{j=1}^m\right) \wedge \neg S_i \text{ is UNSAT.}$$

B.7 Proof of Theorem 6.1

Theorem. Whale *is sound. Under fair scheduling, it is complete for Boolean programs.*

Proof. Soundness: We prove soundness using a generalized version of Hoare's rule for recursion:

$$\frac{c \in Y \quad Y \vdash Z \quad \forall y \in Y \cdot \exists z \in Z \cdot z \vdash y}{c}$$

where Y is a set of Hoare triples over call statements and Z is a set of Hoare triples over function bodies. Given an iARG $\mathcal{I}^{\mathcal{A}}(P)$ that is the result of a terminating execution of Whale that did not return UNSAFE, let

$$\begin{aligned} Y &\equiv \{\{G_j\}\,\vec{b} = F_{\sigma(j)}\,(\vec{a})\{S_j\} \mid \mathcal{A}_j \in \mathcal{I}^{\mathcal{A}}(P) \text{ is uncovered or directly covered}\} \\ Z &\equiv \{\{G_i\}\,B_{F_{\sigma(i)}}\,\{S_i\} \mid \mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P) \text{ is uncovered}\} \end{aligned}$$

If $y \in Y$ is a triple from an uncovered ARG, then, by definition of Y and Z, $\exists z \in Z \cdot z \vdash y$. If $y \in Y$ is a triple from a directly covered ARG $\mathcal{A}_i$, then by soundness of the cover relation $z_j \vdash y$, where $z_j \in Z$ is the body triple of the ARG $\mathcal{A}_j$ covering $\mathcal{A}_i$.

By the above rule, $\forall y \in Y \cdot \vdash y$. Therefore $\vdash \{G_1\}\vec{r} = F_1(\vec{p})\{S_1\}$.

Completeness: To prove completeness of fair Whale on Boolean programs, it is sufficient to show that the algorithm cannot construct an infinite justification tree.

We proceed by contradiction. Assume that the algorithm constructed an infinite justification tree. Note that each $\mathcal{A}_i \in \mathcal{I}^{\mathcal{A}}(P)$ has finitely many call-edges, hence the justification tree is finite branching. By König's lemma, it must have an infinite path.

Let $\pi = \mathcal{A}_{i_1}, \ldots$ be this infinite path. Note that all ARGs on π are uncovered (otherwise the path is finite). Since there are finitely many functions, there is an infinite subsequence $\{k_j\}$ of the path such that $\sigma(i_{k_j}) = l$ for all j, for some function F_l. Furthermore, since the number of Boolean formulas over a finite set of variables is finite, we can assume that $G_{i_{k_j}} = G_{i_{k_{j'}}}$ and $S_{i_{k_j}} = S_{i_{k_{j'}}}$ for all j and j'. Note that for any pair (j, j') s.t. $j < j'$, $\mathcal{A}_{i_{k_j}}$ covers $\mathcal{A}_{i_{k_{j'}}}$. Hence, CoverARG is enabled infinitely often. By fairness assumption, CoverARG must be applied at least once on π. This makes π finite, which contradicts the assumption that the justification tree is infinite. □

Index