Scalable Safety Verification of Statechart-*like* Programs*

Kumar Madhukar
Chennai Mathematical Institute

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

*For the entire work carried out by me during my doctoral research, including this thesis, I was fully supported by Tata Consultancy Services Ltd.

Special thanks to

Thesis Committee

Mandayam Srivas (Co-Supervisor)
Chennai Mathematical Institute, Chennai, India

Peter Schrammel (Co-Supervisor)
University of Sussex, Brighton, United Kingdom

Madhavan Mukund
Chennai Mathematical Institute, Chennai, India

R Venkatesh
TCS Research, Pune, India

To my father

Statement of Attribution

This thesis describes the work from the following conference publications:

1. Kumar Madhukar, Mandayam Srivas, Björn Wachter, Daniel Kroening, and Ravindra Metta. Verifying synchronous reactive systems using lazy abstraction. In Wolfgang Nebel and David Atienza, editors, Design, Automation & Test in Europe Conference & Exhibition (DATE 2015), Grenoble, France, March 9-13, 2015, pages 1571-1574. ACM, 2015.
2. Kumar Madhukar, Björn Wachter, Daniel Kroening, Matt Lewis, and Mandayam Srivas. Accelerating invariant generation. In Roope Kaivola and Thomas Wahl, editors, Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015., pages 105-111. IEEE, 2015.
3. Kumar Madhukar, Peter Schrammel, and Mandayam Srivas. Compositional safety refutation techniques. In Deepak D'Souza and K Narayan Kumar, editors, Automated Technology for Verification and Analysis – 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Springer LNCS Volume 10482, pages 164-183. Springer, 2017.

I was the lead investigator for the work in the first two. These studies were conceived and developed jointly by Björn Wachter, Mandayam Srivas and myself. The implementation and experiments for both (1) and (2) was carried out by me, with significant guidance from Björn Wachter, in a framework which was developed by Björn Wachter and Daniel Kroening. I was primarily responsible for the writing of the manuscripts, to which all other authors made significant contributions.

The work in the third paper was carried out after Peter Schrammel joined my doctoral committee as co-supervisor. The ideas, implementation and

experiments for this work were developed and carried out jointly by Peter Schrammel and myself. The framework in which we implemented our ideas was developed by Peter Schrammel and Daniel Kroening. Mandayam Srivas helped in refining the ideas through several discussions. Peter Schrammel led the writing of the manuscripts, to which Mandayam Srivas and I contributed significantly.

This thesis also contains some unpublished work that was carried out towards the end of my doctoral research. The results are due for submission at a suitable venue in the next few months.

Kumar Madhukar

Statement of Originality

I declare that this dissertation, titled "Scalable Safety Verification of Statechart-*like* Programs", submitted by me for the degree of Doctor of Philosophy in Computer Science is the record of academic work carried out by me during the period of August 2012 to November 2018, under the guidance of Mandayam Srivas and Peter Schrammel, and this work has not formed the basis for the award of any degree, diploma, associateship, fellowship or other titles in this or any other university or institution of higher learning.

26 November 2018 **Kumar Madhukar**

Certificate

I declare that this dissertation, titled "Scalable Safety Verification of Statechart-*like* Programs", submitted by Kumar Madhukar to Chennai Mathematical Institute (CMI), for the degree of Doctor of Philosophy in Computer Science, is a record of bona fide research work done during the period of August 2012 to November 2018, under our guidance and supervision. The work presented in this thesis has not formed the basis for the award of any degree, diploma, associateship, fellowship or other titles at CMI or any other university or institution of higher learning.

It is further certified that the thesis represents independent work by the candidate and collaboration when existed was necessitated by the nature and scope of the problems dealt with therein.

26 November 2018

Mandayam Srivas

(Professor, Chennai Mathematical Institute)

Peter Schrammel

(Professor, University of Sussex, UK,

& Co-founder and CTO, Diffblue)

Abstract

Statecharts extend conventional state-transition diagrams with the notion of hierarchy, concurrency and communication. Their expressiveness makes them a popular language for modeling synchronous reactive systems, a crucial step towards design and development of embedded software for real-life systems. In many instances the system being developed happens to be safety-critical and formal verification of such systems, before their deployment, becomes imperative.

In theory, it is possible to verify a synchronous system using model-checkers that are not tuned for synchronization. The synchrony has to be encoded as part of the input design in such a case. This can be done by either adding a scheduler explicitly, so that the processes are given turns to execute, or implicitly, wherein each process *busy-waits* to synchronize with others. While this approach leverages the power of existing model-checking tools, it is prohibitively inefficient because it fails to exploit the parallelism and control structure inherent to the input system.

We propose an extension of lazy abstraction based model-checking technique, to automatically verify reactive synchronous systems, with the purpose of analyzing synchronous concurrency explicitly rather than encoding it. This circumvents the exponential blow-up of the state space caused by simulating synchronous behaviour using interleavings. SYMPARA, a tool that implements the proposed extension, manifests this state space reduction as an order of magnitude decrease in the time taken for verification over alternative approaches.

A notable advantage of this approach is that it effectively addresses the synchrony and concurrency of systems, without interfering with the core model checking technique. Consequently, one may lift these specialization restrictions on input design, or implementation, and look to improve the verification algorithm itself, with the assurance of a cascade effect to synchronous systems. Pursuing this ahead, we explore ways to generate

program invariants quicker. The state-of-the-art invariant generation techniques, in practice, often struggle to find concise loop invariants. As a result, model checking tools implementing these techniques degrade into unrolling loops, which is ineffective for non-trivial programs. We evaluate experimentally whether loop accelerators enable existing program analysis algorithms to discover loop invariants more reliably and more efficiently. We confirm this empirically through a comprehensive study over a number of benchmarks from the literature.

As acceleration harnesses a low-level control structure i.e. loops, it becomes natural to ask if the higher-level anatomy of an input system offers anything for exploitation, e.g. its modularity. In fact, considering procedures as modules, we look at the problem of finding safety-violations of sequential programs in a compositional way. In this direction, we formalize a space of property-guided compositional refutation techniques, discuss their properties with respect to efficiency and completeness, and evaluate them experimentally.

Towards completion, we build upon our work on compositional refutation to proving safety using k-induction. k-induction is a well-known technique for proving programs safe. However, for large programs, the bounded model checking instances generated to check k-inductive proofs often exceed the limits of resources available. We propose an interprocedural approach to modular k-induction, as an instance of a more general refinement approach to program verification.

Acknowledgments

I would like to express my deep gratitude to my supervisors, Mandayam Srivas and Peter Schrammel for their patient guidance, enthusiastic encouragement and useful critiques in this work. Srivas gave me a lot of freedom to explore my ideas, and always assisted me in enhancing them both theoretically as well as practically. He was extremely supportive and very approachable – qualities that I believe are rare to find in a supervisor. Even though I was staying in a different city from him, he made sure that he was always available when I needed him, not only technically, but otherwise too.

Although it was only in the latter part of my doctoral research that I started interacting with Peter Schrammel, his methods have inspired me a great deal. That it is possible to stay motivated so effortlessly, and work tirelessly towards one's goal, are things that I have known because of him. Peter's ability to sketch the bigger picture at one moment, and then get into the finest of details at the very next moment, is one of a thousand things that there are to learn from him.

I wouldn't have made much progress in the initial years of my PhD, if it was not with the help of Björn Wachter. Björn often went out of his way to offer diligent guidance and valuable advice to me. He helped me clarify my doubts in numerous discussion over calls and ceaseless exchange of emails, when no doubt he had more important things to do.

I would take this opportunity to thank Madhavan and Venky especially. It is not just to acknowledge their guidance and support for this thesis; if I could put an acknowledgement before anything I did that was worthy of credit, these two names would appear prominently in each one of them. Interactions with Madhavan, in the last fifteen years, have moulded me in innumerable ways. Discussions with Venky have been invaluable, and I was blessed to find him around whenever I needed to have one. No doubt many of his insights can be found interspersed in the pages of this thesis.

If not for these two people, this thesis would have never become a reality. The gratitude that I owe them is beyond what words can capture.

I would like to thank Daniel Kroening and the entire Computer Science Department at the University of Oxford, for allowing me to spend some very gainful time there. It was a great learning experience to be a part of their group, notwithstanding the extremely short duration for which I could physically be there. I would also like to thank my friends and colleagues at CMI, for being the great company that they were. Even though I was not a resident student at CMI, I never felt like an outsider whenever I was there.

I consider myself privileged to have been a part of TRDDC during my PhD. I cannot imagine a better place to have spent these years of my life. My friends and colleagues in the VnV group have helped me not only technically, but also personally whenever I needed them. I can't thank them enough for this.

Lastly, I would like to thank my family and friends for their immense love and support.

Kumar Madhukar

Contents

Chapter 1

Introduction

Software systems are ubiquitous; but at the same time, developing good software is hard. There are a thousand opportunities to make mistakes. More importantly, it is difficult if not impossible to anticipate all the situations that a software program will be faced with, more so when it is interacting with other software programs that are not under one's control. In other words, exhaustive testing is impracticable due to the size of the input space. A feasible alternative is that of scenarios based testing. But not all scenarios may be known. Besides, it is possible that certain cases get overlooked, or one may fail to consider the corner cases. Therefore it is essential, especially in safety-critical systems, that a software system is proved correct before it is deployed.

Model Checking is a formal verification technique which allows for desired behavioural properties of a given system to be verified. It requires a model of the system under consideration (for instance, an implementation), and a desired property, and systematically checks whether or not the given model satisfies this property. The check may output *yes* (if there is a *proof* that the property indeed holds), *no* (in case there is a *counterexample*, i.e. an execution path violating the property), or *unknown* (when the check exhausts the physical limits of computer memory available).

Proving correctness of software is one of the most central challenges in computer science. Modern state-of-the-art verification techniques still fail to scale to large, real-life systems. Our work is aimed at addressing this problem, under the principal thesis that *it is useful to exploit the source-level syntactic and semantic structure of a given design, or an implementation, for improving the scalability of safety verification.*

1.1 Motivation

While automated verification techniques have progressed significantly in the last several years, scalable software verification continues to be a challenge. On real-life instances,

the existing approaches often exceed the limits of resources available for a given verification task. We consider *safety* properties, which assert that the system always stays within some allowed region in which nothing "bad" happens. We investigate methods to scale verification of safety properties, with the aim of exploiting the source level syntactic and semantic structure of a software system.

The initial motivation for this work comes from the desire to verify Electronic Control Units (ECUs) in the automotive domain. We consider Statecharts [60], the design language in which these ECUs are given, and explore the possibilities of formally verifying a rather general system in this language. We address the scalability issue arising out of three important aspects of automotive ECUs designed as Statecharts: i) *concurrency*, ii) *loops*, and iii) *scale* (or, in other words, the size of the input software to be verified). In line with this, there are three major parts of our work, each focusing on an aspect listed here. In the following section we lay the context in which we study these aspects, accurately state the problems, and briefly discuss why these problems are of interest.

1.2 Context, Problems, and Objectives

We begin our study with Statecharts, a visual formalism proposed by David Harel, and restrict ourselves to the STATEMATE semantics of Statecharts [61]. Statecharts are popular as a language for modeling synchronous reactive systems, although the expressiveness complicates the task of formally verifying these systems. In the first part of our work we explore the *concurrency* aspect of this problem. In particular, we look at the issue of redundant interleavings, brought about by the synchronous concurrency of Statecharts under STATEMATE semantics.

Existing approaches to formal verification of Statecharts (or, in fact, most other formalisms specifying synchronous reactive systems) predominantly rely on extracting the underlying global transition relation for the system [16, 76, 105]. This approach permits the application of a wide range of standard methods for state space exploration such as BDD-based Model Checking, Bounded Model Checking, k-induction or interpolation, but has the disadvantage of requiring to explicitly construct a scheduler to account for all possible process interleavings, thereby increasing the complexity of verification substantially [30, 93].

An alternative is to use a verifier for asynchronously composed threads (e.g. [104]) and instrument the model to enforce synchronization of the processes at their synchronization points. This not only adds further states per thread (owing to the

synchronization), but also leads to exploration of many irrelevant states, as interleavings that happen between the synchronization points have no effect.

A crucial limitation of these approaches lies in their failure in harnessing the parallelism and control structure inherent to the system's implementation. In accordance with our thesis, we attempt to exploit the semantics of STATEMATE Statecharts to ease the task of verification. Specifically, we answer the following question:

- *For synchronous reactive systems modeled as* STATEMATE *Statecharts, is it possible to analyze synchronous concurrency explicitly, rather than encoding it as part of the system's implementation?*

We suggest an extension of lazy abstraction based model-checking technique for automatic verification of synchronous reactive systems [80] to meet the set objective. Moving further, in the second part of this work we focus on another aspect of such systems - *loops*. While the motivation for this comes from Statecharts, loops are very much a part of any non-trivial system irrespective of the implementation language. For broader applicability therefore, we look at this aspect in the context of sequential programs. As concurrent programs have sequential components, a technique for sequential programs would be imminently extensible to concurrent programs.

The state-of-the-art invariant generation techniques, in practice, often struggle to find concise loop invariants. Consequently, model checking tools implementing these techniques degrade into unrolling loops, which is ineffective for non-trivial programs. Loop unrollings help in obtaining stronger invariants progressively. As an example, for tools inferring invariants using interpolation, unrollings help in moving from overly specific interpolants, with respect to a given unrolling, to more general ones. At the same time, even for tools that do not compute an invariant explicitly, like CBMC [31], unrollings help in the implicit discovery of invariants that strengthen the premise, so that the goals may be discharged.

With the objective of generating program invariants quicker, we explore the option of loop acceleration [18,19,46,65,68] as a precursor to verification. Acceleration captures the exact effect of arbitrarily many iterations of an integer relation, by computing its reflexive transitive closure in one step. A loop accelerator, thus computed, can replace the loop in the original program without impacting its behaviour. In this context, we specifically address the following question:

- *Do loop accelerators enable existing program analysis algorithms to discover loop invariants more reliably and more efficiently?*

Acceleration in the general case is as difficult as the original verification problem. Practical applications of acceleration are therefore usually restricted to some special cases. As the transitive closure of the loop body is often not effectively computable, it is in general not possible to obtain an accelerator that captures the behaviour of the loop precisely. Thus, acceleration is mostly either over-approximative or under-approximative [18, 68, 73]. Besides, acceleration frequently specializes in particular application domains, e.g., control software [1, 43]. Acceleration techniques are also typically tuned to a given analysis technique (e.g., abstract interpretation or predicate abstraction [39,55]) that is to be applied subsequently. As a result, it is not immediately evident if the question posed above can be answered rather generally.

We perform an extensive experimental evaluation [81] and provide strong empirical evidence to answer the question in the affirmative. This brings us to the final part of our work where we attend to the issue of *scale*. We aim at exploiting modularity, a quality intrinsic to large software systems. Specifically, we investigate the following to begin with:

- *Is it possible to generate counterexamples in a modular way to speed up safety refutation?*

Refutation algorithms are usually based on finding a violating execution trace, which seems to be inherently non-compositional. Thus, the study of compositional refutation is an under-explored area of research. Yet, solutions to this problem have significant impact on other research problems. As a motivation, consider the following two algorithmic approaches in verification and testing that will be enabled by efficient compositional refutation algorithms:

- Property-guided abstraction refinement algorithms need to decide whether counterexamples that are found in the abstraction are spurious or true counterexamples. The lack of compositional refutation techniques forces these algorithms to operate in a monolithic manner and is therefore an obstacle to scaling them to large programs.

- Automated test generation techniques based on Bounded Model Checking are successfully used in various industries to generate unit tests. However, they do not sufficiently scale to accomplish the task of generating integration tests. Compositional refutation techniques achieve exactly this goal: they efficiently produce refutations (from which test vectors can be derived) on unit (module)

level and enable their composition in order to obtain system level refutations, i.e. integration tests.

As a first step in this direction, we lay the base for a systematic study of the problem domain. We start from the monolithic safety verification problem and present a step-by-step derivation of the compositional safety refutation problem. We also propose three techniques for compositional safety refutation with different degrees of completeness [79].

Having looked at refutations, we naturally turn towards the problem of safety proofs, exploring if these proofs can be constructed in a modular way. We use k-induction, a popular techniques for proving safety, and formulate the verification problem using k-induction within a generic refinement algorithm framework. In particular, we ask the following question:

- *Is there a compositional approach to k-induction, in an abstraction-refinement framework that allows a component-wise refinement instead of a monolithic one?*

Analyzing code as a monolithic, usually flattened, entity instead of using a divide-and-conquer approach prevents one from exploiting the syntactic and semantic structure, e.g. procedure hierarchy, in the program. Bounded model checking, for instance, unfolds and inlines procedures in the program while unwinding the loops. Most leading complete model-checking methods for safety verification, including ones that use over-approximations, are not compositional [20, 82].

Building upon our work on safety refutation, we propose an interprocedural k-induction approach based on an interprocedural counterexample spuriousness check and a selective refinement of loop unwindings and procedure inlining. In the next section we summarize our results, and highlight the core contributions made in this piece of work.

1.3 Contributions

In conformity with the problem and objectives illustrated in the previous section, we present the contributions made through our work in the following subsections.

1.3.1 Verifying Synchronous Reactive Systems

We explore a novel way of verifying synchronous reactive systems [80] that uses a model checking algorithm based on Lazy Abstraction with Interpolants (LAwI) [83], also known as the Impact algorithm. This algorithm unwinds the control-flow graph of the program into an *abstract reachability tree*. Each vertex in this tree corresponds to a program control location, and is labeled with a fact about the program variables that is true at that point in the execution of the program. When a vertex corresponding to the error location is reached, the algorithm strengthens the facts along the path to that vertex, so as to prove the error vertex unreachable. The crux of the algorithm is a *covering* criterion that allows it to soundly stop the unwinding and terminate with a correctness proof of the program. This combination of low-cost program unwindings combined with path-based refinement and covering checks gives rise to an efficient software model checking algorithm.

Recently, the Impact algorithm was extended to support asynchronous concurrent processes using an interleaved semantics and implemented in a tool called Impara [104]. Impara, which analyses concurrent C programs with POSIX or Win32 threads, combines partial-order reduction with the Impact algorithm. We have tailored and optimized Impara to implement our new technique, which we call Sympara.

Specifically, we make the following contributions through this work:

- A novel *concurrent static-single assignment (SSA)* form, which enables using a fixed schedule for process execution in synchronously composed systems.
- A *covering criterion* that dictates when the unwinding must stop, while ensuring correctness of the algorithm in the synchronous setting.

  This, in fact, relates to our main result:

  Theorem. *The proposed covering criterion is sound, i.e. the unwinding stops only when either a counterexample is found, or the error location has been proved unreachable.*

- A tool, Sympara, that implements this technique in the Cprover [31] framework, and experimental results that show an order of magnitude improvement over other techniques for several realistic examples.

1.3.2 Loop Acceleration as a Precursor to Verification

In this work, we evaluate experimentally whether loop accelerators enable existing program analysis algorithms to discover loop invariants more reliably and more efficiently. We do this through a comprehensive study on the synergies between acceleration and invariant generation.

We use acceleration to summarize loops by computing a closed-form representation of the loop behaviour [72]. The closed form can be turned into an accelerator, which is a code snippet that skips over intermediate states of the loop to the end of the loop in a single step.

We make the following conjectures:

1. Accelerators support the invariant synthesis that is performed by program analyzers, irrespective of the underlying analysis approach.

2. Analyses supported by acceleration not only do better than the original ones, they also outperform other state-of-the-art tools performing similar analysis.

We test our hypotheses by performing an evaluation over an extensive set of benchmarks and a variety of tools. The core contribution of this work is an *experimental study* [81].

We use two analyzers in our experiments to substantiate the first claim (that accelerators aid existing analyzers). CBMC [31] is the model checker used in [74]; as a bounded analyzer, it makes no attempt to infer invariants and is only able to conclude correctness if the program is shallow. IMPARA [104] is a program verifier based on the LAWI-paradigm. IMPARA generates invariants using a very basic approach that relies on weakest preconditions, and does not employ any powerful interpolation engine.

Both IMPARA and CBMC are characterized by very weak invariant inference, and are thus expected to benefit substantially from acceleration. To relate the outcome to the best invariant generation techniques, towards validating our second claim, we include two other analyzers: CPAchecker [13] and UFO [3]. These tools implement a broad range of invariant generation methods, including various abstract domains and interpolation. The comparison is performed on over 200 benchmarks, including those used in the Software Verification Competition 2015 [11].

Acceleration enabled CBMC to handle 21% more cases, and IMPARA to handle 8% more cases. The number of benchmarks correctly proved safe increased by 65% and 10%, respectively, for CBMC and IMPARA. For benchmarks correctly proved unsafe, acceleration brought about a rise of 28% for CBMC and 30% for IMPARA. The

experiments also demonstrated that these tools, when combined with acceleration, outperformed UFO and CPAchecker in terms of both safe and unsafe instances solved correctly (cf. Table 3.1).

1.3.3 Exploiting Modularity of Implementation: Refutations

As a first step towards exploiting the modularity of an implementation, we investigate compositional refutation techniques in hierarchical, e.g. procedure-modular, decomposition of sequential programs. Unlike verification, refutation algorithms are usually based on finding a violating execution trace, which seems to be inherently non-compositional. As a result, the compositional refutation problem is not very well studied. Through our work [79], we lay the base for a more systematic study of this problem domain. Specifically, we make the following contributions:

- To place the problem in a wider context, we give an informal overview on how completeness relates to problem decomposition in safety refutation and verification.
- We formalize the problem space of safety refutation in *hierarchical decomposition* and characterize the compositional completeness guarantees of various algorithmic approaches.

  Our main result, in decomposing the refutation problem, states that:

  Theorem. *For finite state programs, the proposed compositional refutation algorithm with respect to hierarchical (e.g. procedure-modular) decomposition is sound and complete.*

- We propose three refutation approaches, *Concrete Backward Interpretation*, *Abstract Backward Interpretation*, and *Symbolic Backward Interpretation*, in increasing order of completeness.
- We provide an implementation of these safety refutation techniques as an extension to 2LS [20, 96], a verification tool built on the CPROVER framework, and give experimental results comparing their completeness and efficiency.

1.3.4 Exploiting Modularity of Implementation: Proofs

In order to exploit the modularity of implementations for safety proofs, we extend our work on safety refutation to modular safety verification using k-induction [100]. k-induction is one of the most popular techniques for proving safety. However, for large programs, the bounded model checking instances generated to check k-inductive proofs often exceed the limits of resources available. In order to address this:

- We formulate verification by k-induction within a generic refinement algorithm framework.
- We propose an interprocedural k-induction approach based on an interprocedural counterexample spuriousness check and a selective refinement of loop unwindings and procedure inlining.

There are two important results that we establish in this context:

Theorem.
i) If monolithic k-induction proves a property for some finite k then the interprocedural k-induction algorithm will prove it.
ii) If monolithic k-induction finds a counterexample after a finite k number of iterations, then the interprocedural k-induction algorithm will find it too.

1.4 Thesis Outline

The rest of this thesis is organized as follows. Each subsection of Section 1.3 has been expanded into one chapter, that states, motivates and presents our solutions to the problems discussed above. The results shown in Chapters 2-4 have been published and presented at different peer-reviewed conferences, while those in Chapter 5 are still unpublished.

The preliminaries needed for each chapter are discussed in that chapter itself, or in an earlier chapter (and referenced fittingly). The contributions made through different pieces of work, that this thesis incorporates, are highlighted towards the beginning of every chapter.

Chapters 2, 3, and 4 have very minimal dependency on one another, and may in fact be read in any order. Chapter 4, however, is a prerequisite for Chapter 5. Chapter 6 concludes this thesis, and lists interesting directions of future work.

Chapter 2

Verifying Synchronous Reactive Systems

In this chapter we look at Statecharts, a visual formalism proposed by David Harel, and interpret them according to STATEMATE semantics of Statecharts [61]. Statecharts are popular as a language for modeling synchronous reactive systems, although the expressiveness complicates the task of formally verifying these systems. We focus on the issue of redundant interleavings, brought about by the synchronous concurrency of Statecharts under STATEMATE semantics.

2.1 Synchronous Reactive Systems and Lazy Abstraction

Synchronous reactive processes are widely used for modeling and model-based design of embedded software systems. Processes of this kind synchronize at designated points in their control flow. Harel's *Statecharts* [60] is a popular formalism for specifying such processes. Statecharts extend conventional state-transition diagrams with the notion of hierarchy, concurrency and communication. Since we are not concerned about how the synchronizing components actually execute on a machine, we take the liberty to use the terms *processes*, *threads* and *components* interchangeably in this chapter.

Existing approaches to formal verification of Statecharts (or, in fact, most other formalisms specifying synchronous reactive systems) predominantly rely on building a global transition relation for the system. This permits the application of a wide range of standard methods for state space exploration such as BDD-based Model Checking, Bounded Model Checking, and k-induction [27, 103, 106]. The key disadvantage of the approach is that it requires that a scheduler is added to the model to account for all possible process interleavings [84], thereby increasing the complexity of the model

substantially. Furthermore, the approach fails to exploit the structure inherent in the control-flow graph of the processes.

An alternative may be to use a verifier for asynchronously composed threads and instrument the model to enforce synchronization of the processes at their synchronization points. But not only would this add further states per thread (owing to the synchronization), it will also lead to exploration of many irrelevant states, as interleavings that happen between the synchronization points have no effect.

In this work, we explore a third, novel way of verifying synchronous reactive systems. Our approach uses a model checking algorithm based on Lazy Abstraction with Interpolants (LAwI) [83], also known as the IMPACT algorithm. It unwinds the control-flow graph of the program into an *abstract reachability tree* (ART). Whenever the exploration arrives at an error state, the nodes on the error path are annotated with invariants that prove infeasibility of the error path. The crux of the algorithm is a *covering* check that allows it to soundly stop the unwinding and terminate with a correctness proof of the program. This combination of low-cost program unwindings combined with path-based refinement and covering checks gives rise to an efficient software model checking algorithm.

Recently, the IMPACT algorithm was extended to support asynchronous concurrent processes using an interleaved semantics and implemented in a tool called IMPARA [104]. IMPARA, which analyses concurrent C programs with POSIX or Win32 threads, combines partial-order reduction with the IMPACT algorithm. We have tailored and optimized IMPARA to implement our new technique, which we call SYMPARA. The essence of our algorithm is a novel *concurrent static-single assignment (SSA)* form which enables us to use a fixed schedule for process execution.

While the motivation for this work has been to formally verify STATEMATE [61] Statechart specifications, some of our ideas are imminently applicable to other similar formalisms like Esterel [10], Lustre [26], and SIGNAL [51].

2.1.1 Contributions

We summarise the contributions of this chapter as follows.

- We present an extension to the IMPACT algorithm to handle concurrent processes as per STATEMATE semantics. Specifically, the new algorithm supports reactive generation of external events, blocking and non-blocking assignments, and synchronous scheduling of concurrent processes with checks for non-determinism

and race conditions. We prove that our extension, including a change to the criteria for covering, is sound and complete (Section 2.5).

- We have implemented the new algorithm in a tool, called SYMPARA, with a C front-end. We have verified several examples derived from real Statechart models in SYMPARA, IMPARA, and CBMC to compare their relative performances on same designs. Our experiments confirm that SYMPARA performs significantly better compared to other two approaches (Section 2.6).

2.2 Discussion on Related Work

There have been several attempts at applying formal verification to Statecharts. [15] gives a good survey of this topic. Most of these efforts [16, 34, 76, 105] have used the global transition approach, i.e. they involve extracting the global transition relation implemented by the Statechart system and coding it in the logic/language of a popular model checker, e.g. SMV [25], VIS [24], and SPIN [66].

SYMPARA differs from the existing approaches by meeting the following objectives: i) the verification technique underlying SYMPARA is based on LAWI, ii) SYMPARA supports input in ANSI-C, offering applicability to formalisms that permit code generation, and iii) SYMPARA preserves and exploits parallelism, control structure, and hierarchy inherent to Statecharts. The work in [93] comes closest in terms of the last objective. The authors of [93] build a compiler for translating STATEMATE Statecharts into CSP and then verify it using the FDR model checker for CSP [92]. There have also been other attempts at exploiting behavioural hierarchy, e.g. [5, 7]. However, this has only been demonstrated for *hierarchical reactive modules*, a variant of Statecharts.

The most closely related work is by Cimatti et al. [30], who have used a similar software model checking approach based on lazy abstraction to verify SystemC models. They also avoid adding a scheduler, however, their analysis explores all possible interleavings. A distinguishing feature of SYMPARA is its ability to work with a *fixed schedule* of process execution, accounting for synchronous races with an efficient encoding.

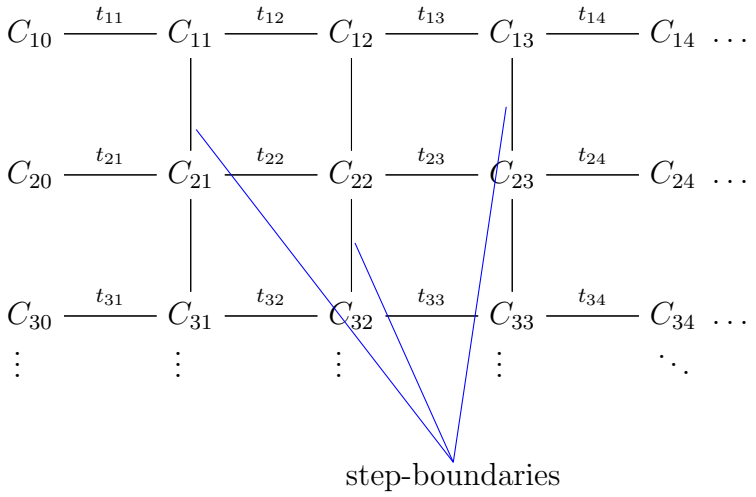


Figure 2.1: Parallel components evolve in steps and synchronize at step-boundaries

2.3 Preliminaries

2.3.1 Statecharts: Syntax and Semantics

Before getting to our technique, we briefly discuss the semantics that we will be using throughout this chapter.

Following STATEMATE conventions, a transition labeled $e[c]/a$ is said to be *enabled* if the *event* e (treated as a predicate for the purpose of this discussion) and *condition* c evaluate to `true`. Here, a denotes the *action* that is performed when the transition occurs. STATEMATE requires execution to progress in *steps*. In each *step*, a transition, if one is enabled, is executed in each process. It is this property of synchronous systems that we exploit in order to fix a schedule. The actions execute atomically and take effect only at the end of that step. Fig. 2.1 shows the evolution of such a system through the steps taken by its parallel components (processes). Each component C_i starts in its initial state, C_{i0}, and evolves by taking an enabled transition in each step. The actions performed by one component are invisible to other components until all the components hit the subsequent step-boundary (the vertical line joining the components C_{*j}, as indicated in Fig. 2.1). It is at these step-boundaries that all the changes, e.g. assignments happening in the actions, take effect. Note that this may give rise to *write-write* races, if conflicting values are set to the same variable by two different components in the preceding step.

The system keeps evolving till it reaches a state where none of the transitions are enabled. Such a state is called a *stable* state. At this stage, inputs from the environment are read and the system evolves further. Evidently, a sound verification

technique for synchronous reactive systems must reason over all schedules, which, in the worst case, requires effort exponential in the number of steps executed by each thread.

In certain circumstances it is possible to guarantee that there is no race, e.g., when the enabled transitions modify disjoint sets of variables. In such a situation it is possible to execute the transitions sequentially, using any ordering. Our key observation is that even in a case of a race, it is possible to encode different behaviours, while still working with a fixed execution order. We illustrate on this observation later in this section, after an overview of our translation scheme.

2.3.2 Lazy Abstraction with Interpolants

Lazy Abstraction with Interpolants (LAwI) [83], also known as the Impact algorithm, is one of the most efficient algorithms for addressing the data state explosion problem for verification of sequential programs. The algorithm returns either a safety invariant for a given program, finds a counterexample or diverges (the verification problem is undecidable).

To this end, it constructs an abstraction of the program execution in the form of a tree, while annotating nodes with invariants. Essentially, the algorithm unwinds the control-flow graph of the program into an abstract reachability tree (ART). Each vertex in the tree corresponds to a program control location, and is labeled with a fact about the program variables that is true at that point in the execution of the program. Each vertex is initially labeled *true*. When a vertex corresponding to the error location is reached, the algorithm strengthens the facts along the path to that vertex, so as to prove the error vertex unreachable.

This strengthening is done by generating an interpolant for the path to the error state. An interpolant for a path is a sequence of formulas assigned to the vertices of the path, such that each formula implies the next after executing the intervening program operation, and such that the initial vertex is labeled *true* and the final vertex *false*. Interpolants can be derived from the unsatisfiability proofs of infeasibility of paths, and, therefore, existence of an interpolant implies that the error vertex is unreachable.

To put abstract reachability trees to work for proving program correctness for unbounded executions, a criterion is needed to prune the tree without missing any error paths. This role is assumed by a *covering* relation between the nodes. Intuitively, the purpose of node labels (denoted by ϕ) is to represent inductive invariants, i.e., over-approximations of sets of states, and the covering relation (denoted by $\triangleright$) is the equivalent of a subset relation between nodes. Consider two nodes, v and w, which

share the same control location (a vector of individual thread locations, which we denote by $\mathfrak{l}$), and $\phi(v)$ implies $\phi(w)$. If we establish that the superset node w cannot be on an error path, we do not need to search for an error path from subset node v. Therefore, if we can find a safety invariant for w, we do not need to explore successors of v. In this case, we say that the node w *covers* the node v.

2.3.3 Verifying Multi-threaded Software with Impact

Impara [104] extends the original Impact algorithm [83] to concurrent software. It constructs an ART, much like Impact, but iterates over all threads to account for the concurrency. The single control location gets transformed into a vector, and the expand routine enumerates all possible interleavings. This algorithm is very inefficient in its basic form; due to the full interleaving semantics, the number of global control locations grows very quickly. Impara amends this by combining monotonic partial-order reduction (POR) with Impact.

2.4 Our Encoding

2.4.1 Statecharts to Concurrent C Programs

Given a system consisting of several components, we translate each component into a separate process. The transition relation is encoded using a program label for every state and, depending on the guard conditions, a `goto` jump from one program label to another. Concurrency is achieved by creating threads and assigning every process to a thread. Sympara's construct `sync()` creates these threads, assigns processes to them and ensures that every thread, when scheduled, executes exactly one statement of the process assigned to it. Communication is encoded with the help of global variables. Hierarchy may be achieved by spawning new threads inside a thread, to represent the components at the next level of hierarchy. This may lead to multiple levels of nesting. Sympara, by default, forces a context switch after every statement. It does so to correctly capture the synchronous semantics of Statemate, which dictates that every enabled component takes a transition and the variable values get synchronized at the end of such a step. However, to model a different semantics, or when there are complex transitions that are written across multiple statements, atomic sections can be used to control the granularity at which interleavings should be analyzed. The desired granularity differs depending on the modeling language. We use auxiliary (`_shadow`) variables to encode *actions* along a transition, thus preventing them from taking

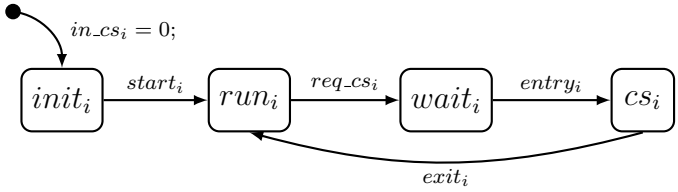


Figure 2.2: State-transition diagram for i^{th} process

effect immediately. The values of these auxiliary variables are copied to the actual variables at the end of each step, when the step-boundary is reached (see Fig. 2.1). The *environment* is modeled as a separate process, which generates non-deterministic values for the environmental inputs.

Let us consider an example consisting of two processes, p_i and p_j, executing concurrently and synchronously, following a mutual exclusion protocol, as shown in Fig. 2.2. The processes begin in an *init* state and then, after receiving the *start* signal, move to a loop where they request access to a critical section, wait till the access is granted, enter the critical section, and finally exit the critical section to get back to the head of the loop. The system evolves through a sequence of steps, where in each step both the processes receive the available signals (which may be thought of as external events, or inputs from the environment) at the same time, and then take an enabled transition together. The actions that happen as part of the enabled transitions do not take effect immediately; they come into effect only after all the processes have completed the step. This is how the processes are synchonously composed in the system.

The variables $start_i$, req_cs_i and rel_cs_i are inputs (events) supplied by the environment. The value of in_cs_i indicates whether p_i is in state cs_i or not. It is set (reset) by all transitions entering (exiting) cs_i. The symbols $entry_i$ and $exit_i$ are place holders for the labels "$[(lock = 0) \wedge (\oplus_k\ in(wait_k))]$ / $lock = 1; in\_cs_i = 1;$" and "rel_cs_i / $lock = 0; in\_cs_i = 0;$", respectively. The variable *lock* is shared and used by the processes to mark their entry to *cs*. The unary predicate $in(s)$ is `true` if the corresponding process is in state s. Fig. 2.3 outlines the translation of this protocol.

2.4.2 Concurrent SSA for Data Races

The static single assignment form (often abbreviated as SSA form or simply SSA) [40] is a way of structuring the intermediate representation so that each variable is assigned exactly once, and every variable is defined before it is used. We introduce the notion of a *concurrent SSA* in this section, and show how it allows us to account for data

```
void p_i(){
  init_i:
  atomic{
    if(start_i){
      /* no actions */
      goto run_i;
    }
    else{
      goto init_i;
    }
  }

  ...

  wait_i:
  atomic{
    if((lock == 0) &&
    (in_wait_j == 0)){
      lock_shadow = 1;
      in_cs_i_shadow = 1;
      goto cs_i;
    }
    else{
      goto wait_i;
    }
  }

  cs_i:
  ...
}
```

```
void environment(){
  atomic{
    /* variable updates */
    lock = lock_shadow;
    in_cs_i =
       in_cs_i_shadow;
    ...
  }
  /* external inputs
     to be generated if
     system is inactive */
  if(inactive){
    /* external inputs */
    start_i = *;
    req_cs_i = *;
    ...
  }
}

void main(){
  atomic{
    sync(): p_i();
    sync(): p_j();
  }
}
```

Figure 2.3: Code snippet for the protocol in Fig. 2.2

races. A data race, in our context, arises when two or more components modify a variable in the same step. Since the effect of transitions take place only at the end of a step, conflicting modifications to a variable by two different threads give rise to a race. In such cases the variable can take a value assigned by any of the threads as the final value. In order to avoid the analysis to branch at this stage, accounting for all possible behaviours, we use concurrent SSAs.

Consider the example in Fig. 2.2, and the code structure corresponding to it (shown in Fig 2.3). We are interested in verifying the correctness of this protocol, i.e. $in(cs_i)$ and $in(cs_j)$ must never evaluate to 1 simultaneously, as per the STATEMATE

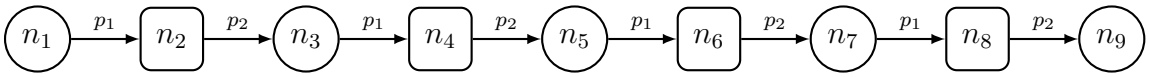


Figure 2.4: An (incomplete) ART corresponding to the example in Fig. 2.2

semantics. Note that if the processes are in states cs_i and cs_j (say), and if rel_cs_i and rel_cs_j evaluate to `true`, then p_i and p_j modify the variable *lock* in the same step. While both the processes here set *lock* to 0, the values may differ in general. Instead of considering both orders of execution (p_i followed by p_j and vice-versa), based on a non-deterministically chosen boolean b, SYMPARA introduces the assignment "$lock = b?v : v'$" where v and v' are the values assigned to *lock* by different threads. We term the SSA generated from such a conditional expression as *concurrent SSA*. It is concurrent SSAs which allow fixing an execution order. Note that concurrent SSAs are needed only when v is not equal to v'. During formal analysis, SYMPARA does syntactic checks and constant propagation before invoking a decision procedure to resolve this (in)equality.

Fig. 2.4 shows the first few nodes of an abstract reachability tree (ART) obtained by unfolding the control-flow graph (CFG) of p_1 and p_2 along a path. The execution begins at node n_1 where both the processes are in their default state. As the unfolding progresses, new nodes are formed. For instance, say n_2 refers to the state where p_1 is in $init_1$ and p_2 is still in its default state, n_3 refers to the state where both p_1 and p_2 are in the *init* state, and so on. Each node has a label associated with it, that captures the invariants discovered so far for the corresponding node. Let n_7 be the node where both the processes are in the *wait* state. Assume the next two steps, to n_8 and n_9, are the $entry_i$ transitions for p_1 and p_2, respectively. This makes n_9 an error node, as both processes are in the *cs* state simultaneously. Once this happens, the algorithm collects the constraints from the initial node, n_1, to the error node, n_9. If the constraints are satisfiable, there is a property violation. In our case, the transition from n_7 to n_8 requires exactly one process to be in the *wait* state, which is not the case at n_7. The node labels are then strengthened to eliminate the path by computing sequential interpolants from the unsatisfiability proof, much like the way it is done in [83] and [104]. SYMPARA terminates when all paths leading to the error node have been eliminated.

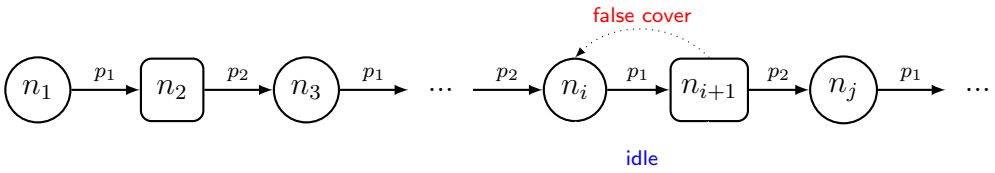


Figure 2.5: The original covering criterion may lead to false cover.

2.5 Technical Details

We now present an extension of the IMPARA algorithm for multi-threaded software, to synchronously composed concurrent programs. Our algorithm, presented in the subsection below, differs from that of IMPARA in primarily two aspects.

- SYMPARA iterates over the set of processes and computes successors in a *fixed* order, tackling races using concurrent SSAs. This fixed order of process execution is encoded in the ART by augmenting the control information at each node with the process scheduled next on that node.
- SYMPARA adds an additional clause in the covering criterion, that the covering node and the covered node must agree on the process scheduled next.

Intuitively, the purpose of node labels (denoted by ϕ) is to represent inductive invariants, i.e. over-approximations of sets of states, and the covering relation (denoted by $\triangleright$) is the equivalent of a subset relation between nodes. Consider nodes v and w that share the same control location (a vector of individual thread locations, denoted by $\mathfrak{l}$), and assume that $\phi(v)$ implies $\phi(w)$. Further, suppose the processes execute in a fixed order and the next process scheduled at v is the same as one scheduled at w. It is easy to see that if there was a feasible error path from v, there would be a feasible error path from w too. Therefore, if we can find a safety invariant for w, we do not need to explore successors of v. In other words, $\phi(v)$ is at least as strong as the already sufficient invariant $\phi(w)$. Hence, if w is safe, so are the nodes in the subtree rooted at v.

In absence of the third criterion (of the next scheduled process being the same), above, if a process p becomes idle (does not change the global system state), the node obtained after executing p could incorrectly be covered by the one before executing p (see Fig. 2.5). The following definition formalizes *cover* for SYMPARA.

Definition 2.5.1. *A node v is said to be covered by another node w (denoted as $w \triangleright v$) if v and w have the same control location, $\phi(v) \rightarrow \phi(w)$ and the next scheduled process on both the nodes are the same, i.e. $sp(v) = sp(w)$.*

2.5.1 The Verification Algorithm

The algorithm (Algorithm 1) constructs an ART by alternating three operation on nodes: Expand, Refine, and Close.

Expand takes an uncovered leaf node and computes its successors along the next scheduled process. The procedure maintains a set of active processes and schedules them in a round-robin fashion to generate new successor nodes. The resulting node is sensitive to the schedule order *only* when there is a race. We use concurrent SSAs to model these races, as described in Section 2.4.2. For every enabled transition, Expand creates a fresh tree node w, schedules the next process on w, updates its location to the target location of the transition and initializes $\phi(w)$ to `true`. The node w is then enqueued to a work list Q and a tree edge is added which records the step from v to w, with the transition constraint R. If w happens to be an error location, the operation Refine is invoked.

Refine takes an error node v, detects if the error path is feasible and, if not, updates the node labels in order to eliminate the path. It determines if the unique path π from the initial node to v is feasible by checking satisfiability of the transition constraints, $\mathcal{F}(\pi)$, along π. If $\mathcal{F}(\pi)$ is satisfiable, the solution gives a counterexample in the form of a concrete error trace, proving the program unsafe. Otherwise, an interpolant is obtained, which is used to refine the labels and update $\triangleright$.

Close takes a node v and checks if v can be added to $\triangleright$. As potential candidates for pairs $w \triangleright v$, it only considers nodes created before v, denoted by the set $V^{\prec v} \subsetneq V$. This ensures a stable behaviour, as covering a node may uncover other nodes. To ensure the soundness of $\triangleright$, all pairs (x, y) where y is a descendant of v, denoted by $v \rightsquigarrow y$, are removed from $\triangleright$ at this point, as v and all its descendants are covered.

Main first initializes the queue with the initial node ϵ, and the relation $\triangleright$ with the empty set. It then runs the main loop of the algorithm until Q is empty, i.e., until the ART is complete, unless an error is found which exits the loop. In the main loop, a node is selected from Q. First, Close is called to try and cover it. If the node is not covered and it is an error node, Refine is called. Finally, the node is expanded, unless it was covered, and evicted from Q.

Theorem 2.5.2. *The modified cover relation never eliminates any permissible behaviours in the synchronous composition.*

Algorithm 1 SYMPARA algorithm for verifying synchronous reactive processes

```
1:  procedure MAIN()
2:      Q := {ε}, ▷ := ∅
3:      while Q ≠ ∅ do
4:          v := dequeue(Q); CLOSE(v)
5:          if v not covered then
6:              if error(v) then
7:                  REFINE(v)
8:              EXPAND(v)
9:      return P is safe
10:
11: procedure CLOSE(v)
12:     for w ∈ V^{≺v} : w uncovered do
13:         if ł(v) = ł(w) ∧ φ(v) ⇒ φ(w) then
14:             if sp(v) = sp(w) then
15:                 ▷ := ▷ ∪ {(v, w)}
16:                 ▷ := ▷ \ {(x, y) ∈ ▷ | v ⇝ y}
17: procedure EXPAND(v)
18:     (ł, φ) := v
19:     T := sp(v)
20:     if T = size(T) then
21:         st(v) = 0
22:         T = 0
23:     for (l, (R, l')) ∈ A(T) with ł_T = l do
24:         // A(T) := actions of T
25:         // R := transition constraint (l → l')
26:         w := fresh node
27:         sp(w) := T + 1
28:         ł(w) := ł[T ↦ l']
29:         φ(w) := true
30:         Q := Q ∪ {w}, V := V ∪ {w}
31:         →:=→ ∪{(v, T, R, w)}
32: procedure REFINE(v)
33:     if φ(v) ≡ False then
34:         return
35:     π := v_0, ... v_N path from ε to v
36:     if F(π) has interpolant A_0 ... A_N
    then
37:         for i = 0 ... N do
38:             φ := A_i
39:             if φ(v_i) ⊭ φ then
40:                 Q := Q ∪ {w | w ▷ v_i}
41:                 ▷ := ▷ \ {(w, v_i) | w ▷ v_i}
42:                 φ(v_i) := φ(v_i) ∧ φ
43:         for w ∈ V s.t. w ⇝ v do
44:             CLOSE(w)
45:     else
46:         abort (program unsafe)
```

Proof. (*Sketch*) Suppose a permissible behaviour does indeed get eliminated from the composition. In other words, the path in the ART corresponding to that behaviour gets prematurely pruned at some node (say, v) during EXPAND, because of the covering criteria. We argue that whenever v gets covered, the node w that covers v can lead to every successor that v could have led to.

Note that the possible successors of a v are entirely determined by its control location, its label $\phi(v)$ and the process scheduled next at v. The coverage criterion ensures that the node w covering v has the same control location and the same process scheduled next. Further, the implication relation between the nodes labels, $\phi(v) \rightarrow \phi(w)$), guarantees that v can only reach a subset of nodes reachable from w. Hence, it is safe to not expand v any further.

Clearly, the missing behaviour is possible from w since all successors of v are reachable from w. In the event that w or one of its successors itself gets covered, we can use a similar argument to establish that the behaviour would still be permissible from the covering node. □

2.5.2 Optimization Strategies

We have added several optimizations in SYMPARA to improve its efficiency. We describe them below.

- SYMPARA handles synchrony in concurrent processes by scheduling every process in each step. Note that there are only two "visible" system states for each step (before any of the threads have executed, and after all the threads have finished execution). However, the implementation allows intermediate states (when only a subset of threads have executed) to be a part of the ART as well (for example, the rectangular nodes in Fig. 2.4). These states are not valid system states, as the synchronous semantics demands that all threads execute simultaneously. However, owing to the fixed schedule of process execution, an intermediate state leads to a unique (non-intermediate) state. Therefore, the effect of covering a state is *eagerly* obtained by covering an intermediate state leading to it.

- SYMPARA tries to replace SMT solver calls by cheaper syntactic checks. First, we use a light-weight decision procedure for cover checks, which decides logical implication in a conservative way, i.e., it finds valid implications but may fail to find an implication that holds. Second, we use syntactic simplification to reduce the complexity of verification conditions that are passed to the SMT solver.

These simplifications may even completely discharge verification conditions in some cases.

- SYMPARA does early elimination of infeasible paths during the path exploration, i.e., the path conditions are decided as they arise, while traditional LAWI lazily decides path-feasibility only when a program assertion is reached. While this scheme of infeasible-path elimination produces a larger number of decision problems, the cascaded approach of cheap checks falling back to SMT, described above, results in an order of magnitude performance improvement.

- SYMPARA resolves potential races eagerly to reduce case-splits. An eager analysis of races allows the option of splitting the problem into doing a race analysis first and, if the system is race-free, avoiding the need to encode the effect of races as conditional assignments to efficiently analyze other properties.

2.6 Implementation and Experiments

We have implemented SYMPARA in the CPROVER [31] framework, the same one in which both CBMC and IMPARA have also been implemented. SYMPARA shares a large part of its code base with IMPARA. As described in the section above, there were three major parts of the implementation: a round-robin iteration strategy for EXPAND, the modification to the covering criterion in COVER, and the optimizations (see Section 2.5.2) in procedures REFINE and EXPAND both. As a backend solver, SYMPARA uses MiniSAT [86].

We experimentally compare SYMPARA with IMPARA and CBMC. Since each of these tools is meant or optimized for a different class of programs, to keep the comparison fair, we have encoded each example used in our experiments in the form that is suitable for the tool it is given to. To obtain a correctness proof in the presence of unbounded control loops, we have used k-induction for our experiments with CBMC.

The benchmarks used in our experiments, listed in Table 2.1, include STATEMATE, Simulink and Lustre programs. The first example, *mutex*, is a simple three-process model for implementing a mutual exclusion protocol. The *vw_alarm* example is a hierarchical eight-process Statechart subsystem extracted from a real model of an alarm system. This example has a large number of conditional transitions from each location that stresses SYMPARA's simplification capability. We have also taken an almost sequential model, *seq_car_alarm*, reverse-engineered from a Simulink Stateflow model. We use this example to quantify SYMPARA's competitiveness on sequential

code. The code has several deeply nested busy-wait timeout loops that are challenging for path exploration. The final three examples, named *dragon*, *switch* and *prod_cons*, have been taken from a set of benchmarks for Lustre (available at `https://bitbucket.org/lememta/lustrebenchmarks`).

Table 2.1 lists the properties checked for these examples. The *correctness* property is with respect to a given specification (for example, at most one process in the critical section, in case of *mutex*). The property of a system to arrive in a stable state within a finite number of steps has been defined as *stability*. The check for *non-determinism* verifies that the system never reaches a state where multiple outgoing transitions are enabled. The properties *independent* and *sensitive* are assertions on program variables, the former independent of the loop in the program, and the latter sensitive to it. All the properties hold for the respective examples, except for *switch*, which is faulty.

Table 2.1 also summarizes our results. Apart from the total run-time (in seconds), we also give the time spent by each tool during SAT solving, and the number of SAT calls for Sympara and Impara. Our experiments were run on a dual-core machine at 2.73 GHz with 2 GB RAM, using a timeout of 900 s. The executables and examples are available at `http://www.cmi.ac.in/~madhukar/sympara/experiments`.

On examples that are either sequential or have a bug (rows 5, 6 and 8), there is little difference between Sympara and Cbmc (the latter performing better in some cases). The benefits are far more significant in all other cases, where Impara and Cbmc fail to generate a proof. The race freedom of Lustre benchmarks also accounts for Sympara's quick convergence in case of *dragon* and *prod_cons* (rows 7 and 9). The unwind column lists the smallest unwinding for which the tool either timed out, generated a proof or produced a counterexample. The reasons for this performance improvement are: (1) Cbmc pessimistically considers all possible schedules, while Sympara works with a fixed one. Even if the system is race-free in all its reachable states, k-induction cannot exploit it as the step-case searches from an arbitrary state. (2) Sympara does well even for cases when the *reachable diameter* (i.e. the maximal distance between two reachable states) is larger than the *initialized diameter* (i.e. the maximal distance from an initial state to a reachable state), owing to its forward search strategy. As compared to Impara, Sympara is efficient as it fixes an interleaving. Another reason for its efficiency is the result of aggressive eager simplifications added in Sympara.

Table 2.1: Experimental Results

	Tools →		**Sympara**			**Impara**			**Cbmc + *k*-Induction**		
No.	**Example**	**Property**	**SAT**		**Time**	**SAT**		**Time**	**Unwind**	**SAT**	**Time**
			#calls	**Time**		**#calls**	**Time**				
1.	*mutex*	*correctness*	632	3.53	5.77	–	–	timeout	69	–	timeout
2.	*mutex*	*stability*	767	5.26	8.02	–	–	timeout	62	–	timeout
3.	*vw_alarm*	*non-determinism*	138	0.64	1.80	–	–	timeout	18	–	timeout
4.	*vw_alarm*	*stability*	1897	87.29	214.82	–	–	timeout	11	–	timeout
5.	*seq_car_alarm*	*sensitive*	202	1.13	1.35	3238	7.23	8.65	2	0.61	2.66
6.	*seq_car_alarm*	*independent*	193	1.12	1.36	4117	7.96	9.96	2	0.15	0.46
7.	*dragon*	*correctness*	115	0.51	0.64	–	–	timeout	65	–	timeout
8.	*switch*	*correctness*	20	0.00	0.02	930	0.26	0.39	3	0.00	0.12
9.	*prod_cons*	*correctness*	30	0.02	0.03	–	–	timeout	1800	–	timeout

2.7 Concluding Remarks

In this chapter we proposed a technique tailored for verifying synchronous reactive systems, as an extension of the LAwI algorithm implemented in Impara. We also described an implementation of our technique into a tool called Sympara. The Ansi-C based input format for Sympara allows formal verification of specifications in a variety of formalisms.

To tune the LAwI algorithm for our context, we introduced several modifications in Sympara. Statecharts tend to have a huge number of case-splits in each transition with a lot of potential for infeasible combination due to delayed assignment semantics. Therefore, we adapted the LAwI algorithm to eliminate infeasible paths early on in the path exploration, i.e. we decide path conditions as they arise, while traditional LAwI lazily decides path feasible only when a program assertion is reached. This scheme of infeasible-path elimination produces a larger number of decision problems that would traditionally all be discharged by an SMT solver, however the cost of such a naive scheme would be prohibitive. Instead, we check if an SMT query, e.g. whether an implication holds, can be resolved by syntactic means, or by using simpler, weaker decision procedures, which ultimately fall back to an SMT solver if their results are inconclusive.

A unique feature of Sympara is that it exploits key features of synchronous parallelism by means of concurrent SSAs, enabling on-the-fly race analysis to prune the search space during symbolic analysis. Our experiments indicate that Sympara provides significant performance advantage over Cbmc, which generates a monolithic constraint corresponding to the global transition relation. Sympara also has the advantage that it is complete, and that it can potentially discover inductive invariants faster than k-induction. When compared to Impara, our experiments suggest that restricting synchrony upfront is far more effective in pruning search space than encoding the same via locks, even with powerful partial-order reduction techniques.

2.7.1 Notes

In order to be widely applicable, Sympara has been developed as an Ansi-C based formal verification tool. This allows support for verification of industry-scale models which are more complex, and often have C code embedded in them. For such complex models, it may be useful to assist Sympara with special invariant generation techniques to efficiently construct node labels. This may help Sympara converge with successful proofs faster and more often.

Loops are one of the most common bottlenecks in scaling verification techniques to complex systems. In the next chapter, we investigate techniques to handle loops, that may be used in conjunction with existing verification methods.

Chapter 3

Loop Acceleration as a Precursor to Verification

A commonly used technique for reasoning about programs with loops is to unroll the loops a finite number of times, such that a) the resulting unrolled program can be analyzed automatically, and b) it can be established that the program never execute an additional iteration of the loops. However, these conditions may be difficult to meet at once for large real-life programs. Therefore, program verification relies on invariants for reasoning about sets of reachable states [48]. A program invariant is a logical assertion that holds during the execution of the program. Similarly, a loop invariant is an assertion that holds on entry into the loop, and is preserved in each iteration of the loop. This chapter experiments with loop acceleration as a precursor to verification, enabling efficient invariant generation.

Acceleration is a technique for summarizing loops by computing a closed-form representation of the loop behaviour. The closed form can be turned into an *accelerator*, which is a code snippet that skips over intermediate states of the loop to the end of the loop in a single step. In this work, we evaluate experimentally whether loop accelerators enable *existing* program analysis algorithm to discover loop invariants more reliably and more efficiently.

3.1 Accelerating Invariant Generation

Consider the program in Fig. 3.1. It contains a simple assertion, which follows the while loop. An automated proof of safety for this assertion requires a technique that is able to discover the loop invariant $(sn = (i - j) * a \wedge (i \leq n)) \vee ((i > n) \wedge (sn = 0))$. State-of-the-art software model checkers either fail to prove the program or even if

they do (for a bounded value of n), they do so by completely unwinding the loop, which does not scale for large n.

```
#define a 2

int main() {
  unsigned int i, j, n, sn = 0;
  j = i;
  while(i < n){
    sn = sn + a;
    i++;
  }
  assert((sn == (n-j)*a) || sn == 0);
}
```

Figure 3.1: Sample Safe Program

The simple recurrent nature of the assignments in the loop of the program makes it amenable to *acceleration* [18,19,46,68]. Acceleration is a technique used to compute the effect of repeated iteration of statements. Specifically, the effect of k loop iterations in the example program is that the variable sn is increased by $k * a$. The idea is to replace, wherever possible, a loop with its closed form to obtain an equivalent accelerated program that is hopefully easier to verify.

Acceleration in the general case is, of course, as difficult as the original verification problem. Practical applications of acceleration are therefore typically restricted to particular special cases. For instance, Jeannet et al. [68] consider the case of deterministic linear loops over continuous variables. As there are very few cases in which the transitive closure is effectively computable, it is frequently not possible to obtain an accelerator that captures the behaviour of the loop precisely. Thus, although acceleration can be precise [9, 46], it is often either over-approximative [18, 68] or under-approximative [73]. Acceleration frequently specialises in particular application domains, e.g., control software. Furthermore, acceleration techniques are frequently tuned to a particular analysis technique (e.g., abstract interpretation or predicate abstraction) that is applied subsequently.

The conjectures that we make are: 1) accelerators support the invariant synthesis that is performed by program analysers, irrespective of the underlying analysis approach, and 2) analysers supported by acceleration not only do better than the original ones, they also outperform other state-of-the-art tools performing similar analysis.

```
int nondet_int();
unsigned nondet_unsigned();

#define a 2

int main(){
  unsigned int i, j, n, sn, k = 0;
  j = i;
  while(i < n){
    if(nondet_int()){ // accelerate
       k = nondet_unsigned(); sn = sn + k*a;
       i = i + k;
       assume(i <= n); } // no overflow
    else{ // original body
       sn = sn + a; i++; }
  }
  assert((sn == (n-j)*a) || sn == 0);
}
```

Figure 3.2: Program from Fig. 3.1 with accelerator

We aim to test these hypotheses by performing an evaluation over an extensive set of benchmarks and a variety of tools.

Since all our benchmarks are C programs, we require an acceleration technique that is applicable to C programs and the fixed-width machine integers that they use. We use a template-based method published at CAV 2013 [73] to obtain the accelerators, and add them to the programs as additional paths. This transformation is exact and thus it preserves safety i.e., the acceleration neither over- nor under-approximates. The implementation of [73] that we use actually works on a goto-binary, a binary representation of the program's control-flow graph as extracted from program source code using `goto-cc` [49]. As a result, we get accelerated goto-binaries instead of real program binaries (say, as `gcc` [37] would produce). Thus, the accelerated programs cannot be given to all common off-the-shelf analysers. Nevertheless, we compare with other tools in our experiments to quantify the advantage that acceleration provides over the state-of-the-art.

Recall our example program. The program with accelerator added is given as Fig. 3.2. The instrumented code in Fig. 3.2 can be used instead of the original code for model checking state properties, as they have equivalent sets of reachable states at the loop heads. We observe that several model checkers that failing on the original

program are able to verify the accelerated program successfully.

The core contribution of this work is an *experimental study*, with the goal to validate our conjectures stated earlier. We quantify the benefit of accelerators when using commodity program analysers. We use two analysers in our experiments to substantiate the first claim (that accelerators aid existing analyzers). CBMC [31] is the model checker used in [74]; as a bounded analyser, it makes no attempt to infer invariants and is only able to conclude correctness if the program is shallow. IMPARA [104] is a C program verifier based on the LAWI-paradigm. IMPARA generates invariants using a very basic approach that relies on weakest preconditions, and does not employ a powerful interpolation engine.

Both IMPARA and CBMC are characterised by very weak invariant inference, and are thus expected to benefit substantially from acceleration. To relate the outcome to the best invariant generation techniques, towards validating our second claim, we include two other analysers: CPAchecker [13] and UFO [3]. These tools implement a broad range of invariant generation methods, including various abstract domains and interpolation. The comparison is performed on over 200 benchmarks, including those used in the Software Verification Competition 2015.

Although acceleration has successfully been combined with interpolation-based invariant construction [65], to the best of our knowledge, there has not been a thorough experimental study that quantifies the benefits of using it in tools that aim to prove correctness. While [73] did integrate acceleration within a framework where paths in the CFG were explored lazily with refinement, the emphasis of their experiments was to accelerate bug detection for unsafe programs. Recently, a loop over-approximation technique based on acceleration was proposed in [41] but this technique is not applicable to unsafe programs. Moreover, there is no refinement to eliminate spurious counterexamples arising from the over-approximation in [41]. The experiments in [74] focus on bounded model checking and do not include state-of-the-art interpolation-based tools.

3.1.1 Contributions

The contribution of this chapter is an extensive experimental study (Sections 3.3 and 3.4) that quantifies the benefit of acceleration when conjoined with off-the-shelf analysis tools.

3.2 Background: Acceleration & Trace Automata

In this section, we go over the preliminaries that includes acceration, for both scalar variables and array assignments, and elimination of redundant paths with the help of trace automata.

3.2.1 Acceleration Overview

The acceleration procedure used in this work is based on the method described in [73]. This method relies on a constraint solver to compute the accelerators. We first provide an overview of the steps of the acceleration procedure, and subsequently provide additional detail. From a high-level perspective, the procedure implements the following steps:

1. Choose a path π through the loop body to be accelerated.

2. Construct a path $\overset{\rightsquigarrow}{\pi}$ whose behaviour under-approximates the effect of repeatedly executing π an arbitrary number of times.

3. The construction also generates conditions under which the acceleration is an under-approximation. These conditions are given in the form of two constraints – a *feasibility constraint*, which denotes the condition under which $\overset{\rightsquigarrow}{\pi}$ can be applied, and a *range constraint*, which constrains the number of iterations. These constraints are included as *assume* statements in $\overset{\rightsquigarrow}{\pi}$.

4. By construction, the assumptions and constraints in $\overset{\rightsquigarrow}{\pi}$ may contain universal quantifiers ranging over an auxiliary variable that encodes the number of loop iterations. The procedure uses a few simple techniques to eliminate these quantifiers that work under certain restrictions. The path is not accelerated if it is not able to eliminate the quantifiers.

5. Augment the control flow graph of the original loop body with an additional branch corresponding to $\overset{\rightsquigarrow}{\pi}$ with a non-deterministic choice in the branch.

6. The accelerated paths subsume some (or sometimes all) paths in the original program. The augmented loop structure generated in the previous step is analyzed to build a trace automaton that filters some of the redundant paths. The result of this step is used to generate a final program with fewer paths.

The acceleration procedure, after executing the above steps, produces an instrumented code with the modifications described in the last two steps. For a program with several loops, possibly nested, the acceleration procedure processes the loops one at a time, inside-out for nested loops. In our experiments we analyse the instrumented code that is produced, without further modifications. This process of acceleration may succeed, fail or time out. The last two outcomes imply that either a closed form solution with a given template does not exist or acceleration was unable to find one.

In the following, we give a few more details of the procedure, the form of the accelerated paths produced and explain the conditions under which the procedure works.

3.2.2 Accelerating Scalar Variables in a Path

For scalar variables, the acceleration is generated by fitting a particular polynomial template. If $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ is the vector of variables in π, then the accelerated assignment generated for each variable $\mathbf{x}$ is represented by the following polynomial function:

$$\begin{aligned} f_{\mathbf{x}}(\mathbf{X}^{\langle 0\rangle}, n) \stackrel{\text{def}}{=} & \sum_{i=1}^{k} \alpha_i \cdot \mathbf{x}_i^{\langle 0\rangle} \\ + & \left(\sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathbf{x}_i^{\langle 0\rangle} + \alpha_{(2\cdot k+1)} \right) \cdot n \\ + & \alpha_{(2\cdot k+2)} \cdot n^2 \end{aligned}$$

Here, n is the number of loop iterations that are summarized, $\mathbf{x}_1^{\langle 0\rangle}, \ldots, \mathbf{x}_k^{\langle 0\rangle}$ are the initial values for the variables and the α_i with $0 \leq i \leq 2k+2$ are the unknown coefficients.

The acceleration for a path is performed in two steps. In the first step, the procedure solves for the coefficients α_i. This is done by considering only the assignments in the path π, i.e., by ignoring all the conditions, including the loop condition. This employs a combination of linear algebra techniques to first uniquely solve for the coefficients and then makes queries to SMT solver to inductively check that the generated polynomial for each variable is consistent with loop execution for an arbitrary number of iterations. If, for some $\mathbf{x}_i$, the inductive check fails, then it means there is no acceleration possible that fits the template for this choice of coefficients.

In the second step, the procedure considers the path with all the conditions, and generates the feasibility constraint, i.e., the condition under which the path is feasible.

In order to guarantee that only states that are reachable in the original program can be reached via accelerated paths, we need to make sure that $\tilde{\pi}$ is only feasible for values of n for which a cumulative path, π^n, is also feasible. We achieve this by computing a pre-condition for $\tilde{\pi}$ that rules out values of n for which π^n is not feasible. The feasibility constraint is essentially the negation of $wlp(\pi^n; \mathit{false})$, where wlp is the weakest liberal precondition. Intuitively, a cumulative path π^n would be infeasible iff any intervening path π in the n-iteration cycle, starting from the state given by the accelerator, is infeasible. That is, π^n is infeasible if for any $j < n$ the first time frame of the suffix $\pi^{(n-j)}$ is infeasible (time frame refers to an instance of π in π^n). Thus, checking whether $wlp(\pi^n, \mathit{false})$ holds is equivalent to checking if, for some j between 0 and n, $wlp(\pi, \mathit{false})$ holds (after substituting every variable in π by its accelerated closed form expressions). Thus, the feasibility constraint for π^n will, in general, contain a universal quantifier ranging over the number of loop iterations. This can be eliminated if the predicate in the body of the formula is monotonic over the quantified parameter. The procedure reduces the monotonicity check in a conservative fashion to a SMT query by defining a *representing function* that returns the size of the set of states for which a predicate is false. No acceleration is performed if the monotonicity check fails.

3.2.3 Range Constraints

Since closed-form expressions and the derived feasibility constraints usually contain the number of iterations n in them, an overflow is likely to break the monotonicity requirement when bit-vectors or modular arithmetic are used. Also, since the behaviour of arithmetic over- or under-flow in C is not specified for signed arithmetic, we conservatively rule out all occurrences thereof in the accelerated path. This is done by adding range constraints in the form of *assume* statements, which enforce that none of the arithmetic expressions that involve n overflow.

3.2.4 Accelerating Array Assignments

Acceleration of array assignments is challenging, as under-approximating closed-form solutions for them can often only be expressed by formulas that contain quantifier alternation (existential inside universal) ranging over the number of loop iterations and the domain (index) of the array. It has been shown in [73] that for array assignments of the form $a[x] := e$ such a quantifier pattern can be eliminated under the following sufficient conditions.

- There exist accelerated closed-form expressions for the index variable x and the expression e.
- The function f_x defining the closed-form solution for the index variable is linear in the number of loop iterations.

Under the above conditions one can derive a closed form representing an under-approximation of the array assignments.

```
int nondet_int();
unsigned nondet_unsigned();

#define a 2
int main(){
  unsigned int i,j, n, sn, k = 0;
  bool g = *;
  j = i;
  while(i < n){
    if(nondet_int()){ // accelerate
       assume(!g);
       k = nondet_unsigned(); sn = sn + k*a;
       i = i + k;
       assume(i <= n); // no overflow
       g = true;}
    else{ // original body
       sn = sn + a; i++;
       g = false;}
  }
  assert((sn == (n-j)*a) || sn == 0);
}
```

Figure 3.3: Program from Fig. 3.2 with instrumented trace automaton

3.2.5 Eliminating Redundant Paths using Trace Automata

The instrumentation of the accelerators described in the introduction preserves the unaccelerated paths in the program along with the newly added accelerated paths – for instance, the *else* branch in Fig. 3.2. Note that the added paths subsume some of the previously existing program paths.

The idea presented in [74] is to eliminate executions that are subsumed by some other execution of the program. For instance, taking the same accelerated path twice

in a row is equivalent to taking it just once (for instance, in Fig. 3.2, executing the *if* block twice for values k_1 and k_2 is the same as executing it once with the value of k equal to $k_1 + k_2$ – which is possible because k is chosen non-deterministically in each iteration).

Similarly, taking the unaccelerated path immediately after taking the accelerated path is subsumed by taking the accelerated path just once (with the value of k being one more than its previously chosen value, in Fig. 3.2). The elimination of these redundant paths is done by encoding the redundancies as a regular expression, which is then translated into a *trace automaton* [63]. When the accelerated program executes, the states in this automaton are also updated and it is ensured that this automata never reaches an *accept* state. Reaching an accept state means that the execution contains redundant iterations of accelerators and, therefore, is not of interest. An optimized version of the accelerated code for the running example is given in Fig. 3.3. This is achieved by introducing an auxiliary variable g that determines whether the accelerator was traversed in the previous iteration of the loop. This flag is reset in the non-accelerated branch, which, however, in our example is infeasible.

3.3 Experimental Setup: Tools & Benchmarks

We start this section with a brief informal introduction of the different tools used for our experiments.

3.3.1 Overview of the Analysis Tools

Ufo [3] combines the efficiency of abstract interpretation with numerical domains with the ability to generalize by means of interpolation in an abstraction refinement loop. Ufo starts by computing an inductive invariant for the given program and checks if the invariant implies the given property. If the implication does not hold, Ufo employs SMT solvers to check the feasibility of the counterexample produced. If the error path is found to be infeasible, an interpolation technique guided by the results of an abstract interpretation is used to strengthen the invariant.

CPAchecker [13] is a tool and framework that aims at easy integration of new verification components. Every abstract domain, together with the corresponding operations, implements the interface of configurable program analysis (CPA). The main algorithm is configurable to perform a reachability analysis on arbitrary combinations of existing CPAs. The framework provides interfaces to SMT solvers and interpolation procedures, such that the CPA operators can be written in a concise and convenient

way. CPAchecker uses MathSAT [29] as an SMT solver, and CSIsat [14] and MathSAT as interpolation procedures. It uses Cbmc as a bit-precise checker for the feasibility of error paths, JavaBDD [67] as the BDD package and provides an interface to an Octagon representation as well.

Cbmc [31] is a bounded model checker for Ansi-C programs. It works by jointly unwinding the transition relation encoded in the given program and its specification, to obtain a first-order formula that is satisfiable if there exists an error trace. The formula is then checked using a SAT or SMT procedure. If the formula is satisfiable, a counterexample is extracted from the satisfying assignment provided by the SAT procedure. The tool also checks that sufficient unwinding is done to ensure that no longer counterexample can exist by means of *unwinding assertions*. This enables Cbmc to prove correctness if the program is shallow.

Impara [104] extends the Impact algorithm to support asynchronous concurrent processes using an interleaved semantics (cf. Section 2.3.3). Impara, which analyses concurrent C programs with POSIX or Win32 threads, efficiently combines partial-order-reduction with the Impact algorithm. This work highlights the benefits of combining Impara with acceleration for sequential programs.

The Impara algorithm returns either a safety invariant for a given program, finds a counterexample or diverges. To this end, it constructs an abstraction of the program execution in the form of an *Abstract Reachability Tree* (ART), which corresponds to an unwinding of the control-flow graph of the program, annotated with invariants. To prove a program correct for unbounded executions, a criterion is needed to prune the ART without missing any error paths. A covering relation assumes this role.

The tool constructs an ART by alternating three different operations on nodes: Expand, Refine, and Close. Expand takes an uncovered leaf node and computes its successors along a randomly chosen thread. Refine takes an error node v, detects whether the error path is feasible and, if not, restores a safe tree labeling. First, it determines whether the unique path π from the initial node to v is feasible by checking satisfiability of the transition constraints along π. If it is satisfiable, the solution gives a counterexample in the form of a concrete error trace, showing that the program is unsafe. Otherwise, an interpolant is obtained, which is used to refine the labels and update the cover relation. Close takes a node v and checks if v can be added to the covering relation. As potential candidates for pairs to be a part of the covering relation, it only considers nodes created before v. This is to ensure a stable behaviour, as covering in arbitrary order may uncover other nodes, which may not terminate.

3.3.2 Benchmarks

We ran our experiments on a set of 201 benchmarks (138 safe, 63 unsafe) collected from the sources listed in [17] (published at CAV 2014) and SV-COMP 2015. We have eliminated examples that had syntax errors and the ones that were not supported by the accelerator (array examples, for instance). We used the accelerator provided by the `goto-instrument` executable that comes with CBMC. When run with the `--accelerate` option, it takes a goto-binary file as input, and produces an accelerated binary corresponding to it. The inputs to the accelerator were generated using the `goto-cc` executable that also comes with CBMC. The accelerator does not work on ANSI-C files directly. Owing to this limitation, we could try only CBMC and IMPARA with acceleration (these tools are built on the same CPROVER framework, and both accept goto-binaries as input). We compare the performance of UFO, CPAchecker, CBMC (with and without acceleration) and IMPARA (with and without acceleration). The unwinding depth used for experiments with CBMC was 100 for unaccelerated programs and 3 for accelerated programs. All experiments were run on a dual-core machine running at 2.73 GHz with 2 GB RAM, with a timeout limit of 60 seconds.

We elaborate on the benchmarks and the tools used to aid reproducibility. The benchmarks were collected from [4, 56, 58], the loops category in SV-COMP 2015 and the acceleration examples in the regression suite of CBMC (revision 4503). The tools used in the experiment were UFO (the SV-COMP 2014 binary), CPAchecker (release 1.3.4, with sv-comp14.properties as the configuration file), CBMC (built from revision 4503, used with Z3 as the decision procedure) and IMPARA (version 0.2, used with MiniSat). The benchmarks, the exact commands used to invoke the tools, and the full results are available at `http://www.cmi.ac.in/~madhukar/fmcad15`.

3.4 Evaluation: Results & Analysis

Before we discuss the results, we present an example to demonstrate the effectiveness of acceleration.

3.4.1 Example

Consider the safe example shown in Fig. 3.4. All the tools involved in our experiments fail to prove this example safe. Even when the timeout is increased to 15 minutes, the tools still timeout. In general, one needs a loop invariant strong enough to prove the assertion outside the loop, to avoid unwinding the loop to the full. None of the

tools were able to find such a loop invariant. Upon acceleration, a closed form for the variable x is generated: $x = 1 * k + 2 * l$, where k and l are the number of times that the *if* and the *else* branches inside the loop are taken. The additional constraint generated for k, that $k = 65520$, along with the closed form for x is sufficient to prove the property.

In some circumstances, acceleration uses quantifiers in the accelerated programs. These are not the ones arising from the feasibility or range constraints that we discussed in Section 3.2 (those get eliminated during the acceleration). These quantifiers appear while encoding the overflow constraints in the accelerated program. Suppose we want to construct a closed form for a variable being modified in a loop, by assuming that the loop executed i times. In this case, we need to assure that there is no overflow that was caused during any of these i iterations. In some cases, it is sufficient to assume that i^H iteration does not lead to an overflow. An instance is example 3.4, as the loop condition is $(x < 268435454)$. Thus, if the i^{th} iteration does not lead to an overflow, none of the previous iterations do. However, if we change the loop condition to $(x \neq 268435454)$ this does not hold any more. Therefore, it must be ensured separately for every $k \in [0, \ldots, i]$ that there is no overflow after k iterations. In our experiments, there were 40 benchmarks (roughly 25 %) that use quantifiers in their corresponding accelerated programs. The presence of quantifiers makes the verification task difficult as none of the tools is able to instantiate the quantifiers correctly. More effective quantifier handling will yield further results in favor of acceleration.

Table 3.1 summarizes the performance of each of the tools. We record the number of safe instances reported as safe (*correct proofs*), the number of safe instances reported as unsafe (*wrong alarms*), the number of unsafe instances reported as unsafe (*correct alarms*), the number of unsafe instances reported as safe (*wrong proofs*), the number of instances which could not be decided by the tool (*no result*), the number of instances on which the tool reported the correct result in the least amount of time (*fastest*), the number of instances on which the tool was the only one to report the correct result (*unique*) and a score for each tool, calculated using the scoring scheme of SV-COMP 2015.[1]

3.4.2 Experimental Results

Impara + Acceleration clearly outperforms Impara without acceleration, Ufo and CPAchecker. This underlines the benefit of acceleration as an auxiliary method for

[1] Score = (2·*correct proofs*) − (12·*wrong proofs*)+*correct alarms*−(6·*wrong alarms*)

```
int main(void) {
  unsigned int x = 0;
  while (x < 268435454) {
    if(x < 65520){
      x++;
    } else {
      x += 2;
    }
  }
  assert(!(x % 2));
}
```

Figure 3.4: A safe benchmark showing the need for acceleration.

Table 3.1: Comparison of tools

Tools	**Number of instances**							**Score**
	correct proofs	wrong proofs	correct alarms	wrong alarms	no results	fastest	unique	
CPAchecker 1.3.4	83	16	35	14	53	18	11	−75
Ufo SV-COMP 2014	52	2	18	2	127	4	2	86
Cbmc r4503	32	0	35	0	134	16	1	99
+ Acceleration	53	0	45	12	91	28	9	79
Impara 0.2	78	1	36	15	71	73	0	90
+ Acceleration	86	0	47	12	56	36	6	147

invariant generation. Note that we see an increase in the number of *correct proofs* as well as *correct alarms*. CPAchecker comes close in terms of the *correct proofs*, which we credit to its broad portfolio of techniques for generating invariants, including interpolation, abstract interpretation and predicate abstraction. The *wrong proofs* CPAchecker generates are partly caused by missing overflow situations.

When compared to Cbmc + Acceleration, Impara + Acceleration does better for the following reason: The accelerators themselves are not helpful to Cbmc for generating proofs – it simply unwinds the program CFG and makes a single decisive query to the solver. A large number of our benchmarks are safe, and Cbmc only benefits from accelerators if the trace automaton is able to prune the original paths. By contrast, even without trace automata, acceleration may improve convergence of Impara, as acceleration can lead to "better" interpolants. Without acceleration an interpolation procedure is presented an unwinding of the loop body. It is well-known, see e.g. [12], that this can lead to overly specific interpolants that rule out only this

```
int main(){
  unsigned int n = nondet_uint();
  int x = n;
  int y = 0;

  // loop invariant: x + y == n
  while(x > 0){
    x = x - 1;
    y = y + 1;
  }
  assert(y == n);
}
```

Figure 3.5: Acceleration can improve generalisation in LAwI.

particular unwinding. By contrast, in the accelerated program, the interpolation procedure is presented with the transitive closure of the loop; it thus is forced to compute an interpolant for a much larger number of unwindings. For instance, Impara without acceleration fails to generate a loop invariant for Fig. 3.5, and thus falls back to loop unwinding, whereas, on the accelerated program, unwinding is avoided, and the tool generates the invariant $x + y = n$.

The overall score drops when combining Cbmc with acceleration. This is due to the wrong alarms generated by the combination, which is heavily penalized according to the scoring rules at SV-COMP. We suspect that this arises from some practical, implementation-specific, limitations of the acceleration method. On the other hand, there is a substantial increase in the number of correct proofs and correct alarms, however. The advantages of combining acceleration with Cbmc and Impara (note that Cbmc and Impara are very different tools) strongly suggests that a similar advantage could be obtained with other tools as well. An investigation of the cause for the increase in number of wrong alarms for Cbmc and a precise quantification of the benefit of combining other tools would be worthwhile directions to explore as future work.

The fact that acceleration helps Cbmc and Impara on unsafe instances is unsurprising; the technique we use was designed to aid counterexample detection [73]. The experimental results confirm that in addition, acceleration helps to generate invariants. Invariant generation techniques, in practice, often struggle to find concise loop invariants, and, instead, degrade into unrolling loops completely, which leads to poor performance and defeats the purpose of invariant generation. Our experiments

demonstrate that there is a synergy between the two techniques. For example, acceleration may provide candidate predicates for an adequate abstraction, or as described above, may simplify the program in a way that more general interpolants are obtained instead of specific ones. That is why we say that acceleration leads to better invariants.

While CPAchecker employs a bit-accurate tool – by default CBMC – to verify counterexamples, its invariant generation engine works over mathematical integers, i.e. invariants may hold over mathematical integers but are not checked with respect to integer overflow. Wrong proofs observed with CPAchecker mainly arise from deriving mathematical-integer invariants that do not hold in presence of overflow. In such situations acceleration cannot help, i.e. though acceleration may help in obtaining an invariant faster, CPAchecker would continue to present wrong proofs unless it accounts for overflows.

Since the accelerator works on goto-binaries, we could not quantify the benefits of acceleration in terms of the number of loops replaced. Table 3.2 gives the complete results of our experiments. If a tool worked on a given benchmark and produced the expected result, we report the time taken by the tool in seconds. The entries to, inc, err and nr indicate, respectively, that the tool timed out, produced an incorrect result, terminated with an error or could not decide whether the input benchmark is safe or unsafe. The winning entry (in terms of the time taken by the tool) for each row (if there is one) is given in bold font. Note that the time taken by CBMC + Acceleration and IMPARA + Acceleration does not include the time taken to generate the instrumented program with accelerators. The latter is given separately in the column *Accl.*

3.5 Concluding Remarks

In this chapter we have quantified the benefit of acceleration for checking safety properties. We report the results of a comprehensive comparison over a number of benchmarks, which shows that the combination of acceleration and a safety checker indeed outperforms existing techniques. The performance enhancement is visible for both safe and unsafe benchmarks, shown by an increase in the number of correct alarms as well as the correct proofs reported by the tool.

Table 3.2: Complete Table of Experimental Results

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
28.c	nr	to	nr	nr	to	to	1.27
25.c	nr	inc	to	**0.69**	to	to	0.52
20.c	nr	1.41	7.01	6.54	**0.02**	0.03	1.17
09.c	nr	to	nr	nr	to	to	1.81
05.c	nr	to	nr	nr	to	to	2.12
f2.c	to	inc	to	40.46	to	**10.17**	1.65
xyz2.c	nr	inc	nr	nr	to	**0.79**	1.14
xy0.c	nr	inc	nr	nr	to	**0.19**	0.53
gulv.c	to	47.18	nr	to	**38.83**	to	3.6
substring1.c	0.35	to	nr	**0.08**	42.79	0.26	0.26
24.c	nr	48.31	to	nr	**3.52**	to	0.49
xy4.c	nr	inc	nr	nr	to	**0.21**	0.68
pldi082_unbounded.c	nr	to	nr	nr	to	to	0.78
15.c	nr	14.65	nr	**0.32**	to	0.64	0.48
gulv_simp.c	**0.57**	46.13	nr	nr	to	to	1.14
33.c	nr	**47.62**	to	to	to	to	2.59
xy10.c	nr	19.78	0.21	0.35	**0.01**	0.02	0.45
xyz.c	nr	inc	nr	nr	to	**17.08**	1.18
12.c	nr	to	nr	nr	to	to	5.59
31.c	nr	inc	to	**1.75**	to	to	0.18
35.c	nr	inc	nr	**0.22**	to	0.35	0.29
07.c	nr	inc	nr	**0.06**	to	0.1	0.26
39.c	0.2	1.22	**0.01**	err	0.02	err	0.5
19.c	nr	inc	nr	**0.62**	to	to	0.83
37.c	nr	inc	nr	**0.24**	to	0.7	0.62
simple_safe1.c	0.21	to	nr	0.06	**0.01**	0.02	0.23
diamond_unsafe2.c	nr	1.45	9.44	**0.4**	0.53	0.84	0.6
underapprox_unsafe1.c	0.2	1.23	0.02	nr	**0.02**	0.03	0.38
nested_safe1.c	**0.2**	to	nr	nr	to	to	0.87
diamond_safe2.c	nr	17.34	2.29	**0.2**	0.43	0.85	0.57
T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl							
nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*							*continued on next page*

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
const_unsafe1.c	0.26	1.5	**0.02**	0.06	0.03	0.03	0.2
functions_safe1.c	0.2	to	nr	0.06	**0.02**	0.03	0.23
nested_unsafe1.c	0.27	to	nr	nr	0.52	**0.09**	0.2
multivar_unsafe1.c	0.32	1.39	0.31	0.1	**0.01**	0.02	0.27
underapprox_safe2.c	0.2	1.39	0.02	nr	**0.02**	0.05	0.37
simple_unsafe1.c	0.2	to	nr	0.07	to	**0.03**	0.23
underapprox_unsafe2.c	0.2	1.49	0.03	nr	**0.02**	0.05	0.39
simple_safe4.c	0.2	to	nr	nr	0.02	**0.02**	0.2
simple_unsafe4.c	to	to	nr	nr	to	to	0.21
simple_unsafe2.c	0.39	1.73	0.17	0.07	**0.01**	0.01	0.22
phases_unsafe1.c	to	to	nr	**0.1**	to	0.63	0.33
diamond_safe1.c	nr	to	0.38	**0.11**	3.99	1.09	0.3
diamond_unsafe1.c	nr	2.49	0.31	0.16	54.33	**0.04**	0.34
simple_safe2.c	0.21	1.53	nr	0.07	**0.01**	0.02	0.19
const_safe1.c	0.21	1.28	**0.01**	0.04	0.02	0.03	0.21
overflow_unsafe1.c	0.2	inc	nr	**0.06**	to	0.11	0.19
simple_unsafe3.c	nr	1.42	0.08	0.08	**0.01**	0.01	0.23
phases_safe1.c	to	to	nr	**0.12**	to	0.93	0.29
simple_safe3.c	0.16	to	nr	0.06	**0.01**	0.03	0.24
multivar_safe1.c	nr	1.54	nr	0.07	**0.01**	0.04	0.28
functions_unsafe1.c	0.2	to	nr	0.06	to	**0.02**	0.22
underapprox_safe1.c	0.19	1.4	0.02	nr	**0.02**	0.04	0.38
overflow_safe1.c	0.21	47.8	nr	0.08	**0.01**	0.03	0.22
efm.c	nr	inc	to	to	inc	err	3.76
hsortprime.c	nr	**51.33**	to	nr	inc	to	1.03
bk-nat.c	nr	inc	25.61	39.74	**0.43**	err	3.38
barbrprime.c	nr	1.78	to	12.41	**0.06**	0.65	5.21
fig1a.c	nr	to	nr	nr	to	to	1.03
swim.c	nr	**48.9**	to	to	to	err	4.2
seesaw.c	nr	**47.73**	to	nr	to	err	1.59

T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl

nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*

continued on next page

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
swim1.c	nr	**52.44**	to	to	inc	inc	4.7
barbr.c	nr	**1.85**	to	22.63	8.38	55.01	7.26
ex1.c	nr	inc	**42.07**	nr	to	to	1.62
cars.c	nr	to	to	to	to	to	13.33
fig2.c	inc	inc	to	24.89	to	**11.54**	1.59
lifo.c	nr	**19.61**	to	to	29.98	40.92	9.43
lifnatprime.c	nr	inc	to	to	inc	**34.76**	8.31
ex2.c	nr	7.07	**0.04**	0.05	8.03	8.41	0.09
bkley.c	nr	inc	23.16	**15.03**	to	err	2.99
hsort.c	nr	inc	to	nr	inc	inc	1.43
lifnat.c	nr	inc	to	to	inc	**58.74**	9.73
seq-len.c	nr	to	to	nr	to	to	1.57
svd-some-loop.c	nr	**2.42**	to	nr	inc	5.74	2.2
split.c	nr	to	nr	nr	to	to	0.12
string_concat-noarr.c	0.21	1.25	0.01	0.18	**0.01**	0.03	0.64
bind_expands_vars2.c	nr	19.08	nr	**1.39**	to	2.42	0.37
simple_if.c	nr	**37.98**	nr	nr	to	to	0.45
nest-if5.c	0.2	19.93	nr	nr	0.01	**0.01**	0.38
up-nested.c	0.21	1.36	nr	nr	**0.01**	0.02	0.25
NetBSD_g_Ctoc.c	0.2	1.22	0.01	0.02	**0.01**	0.02	0.93
nested8.c	nr	**22.81**	to	nr	inc	60.0	1.95
nest-len.c	nr	48.98	to	nr	to	**7.33**	0.76
nested2.c	0.59	47.6	to	nr	**0.01**	0.03	0.27
heapsort3.c	nr	inc	to	nr	inc	inc	0.33
sendmail-close-angle.c	nr	to	nr	**1.2**	to	inc	0.91
NetBSD_glob3_iny.c	0.26	1.26	0.01	err	**0.01**	err	0.55
nested.c	0.2	47.03	to	nr	to	**0.1**	0.24
seq-sim.c	nr	to	to	nr	to	to	1.06
puzzle1.c	nr	1.36	to	nr	0.01	**0.01**	0.63
half.c	nr	to	nr	**0.76**	to	4.28	1.53

T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl

nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*

continued on next page

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
MADWiFi-encode_ie_ok.c	nr	18.98	15.1	0.28	**0.06**	0.14	0.41
simple.c	nr	inc	nr	0.12	**0.02**	0.14	0.15
nested1.c	0.63	47.95	to	nr	**0.01**	0.02	0.25
mergesort.c	nr	1.57	0.02	0.01	0.02	**0.01**	1.1
svd4.c	nr	**3.87**	to	err	inc	inc	6.09
spin.c	0.2	1.26	0.01	0.01	0.01	**0.01**	0.06
svd2.c	nr	1.51	to	nr	**0.01**	0.03	0.74
spin1.c	0.21	1.24	0.01	0.01	0.01	**0.01**	0.07
heapsort.c	nr	inc	to	nr	inc	inc	1.32
nest-if7.c	0.2	48.21	to	nr	**0.09**	0.13	0.48
sendmail-mime7to8_arr_three_chars_no_test_ok.c	0.2	2.75	to	1.58	0.02	**0.01**	0.31
nest-if1.c	nr	inc	to	nr	**0.29**	0.84	0.25
simple_nest.c	nr	1.35	nr	nr	**0.06**	0.29	0.81
NetBSD_loop_int.c	0.2	1.25	0.01	err	**0.01**	err	0.34
nested6.c	nr	47.26	to	nr	**7.88**	8.07	0.5
down.c	nr	to	nr	**0.37**	to	1.3	0.58
seq.c	nr	1.41	7.26	1.98	**0.02**	0.04	1.59
seq3.c	nr	to	to	nr	to	to	1.18
nested3.c	nr	inc	to	nr	to	to	0.39
nest-if2.c	nr	inc	to	nr	20.95	**0.8**	0.39
seq-proc.c	inc	to	to	**2.98**	to	to	1.24
nest-if4.c	0.2	19.91	to	nr	**0.01**	0.03	0.27
apache-escape-absolute.c	nr	to	to	nr	**0.21**	3.34	3.68
bound.c	nr	inc	nr	1.87	**0.02**	0.12	1.76
nest-if.c	0.6	47.82	to	nr	to	**0.13**	0.26
svd3.c	nr	1.58	to	nr	**0.02**	0.03	0.38
up5.c	nr	to	nr	nr	to	to	0.87
heapsort2.c	nr	**1.81**	to	nr	to	to	0.23
NetBSD_loop.c	nr	18.68	4.96	0.21	**0.01**	0.02	0.42
nested9.c	nr	to	to	nr	inc	inc	0.39

T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl

nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*

continued on next page

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
nested4.c	nr	nr	to	nr	to	to	0.38
up3.c	nr	to	nr	nr	to	to	1.03
heapsort1.c	nr	**1.79**	to	nr	to	to	0.25
SpamAssassin-loop.c	nr	18.84	to	err	**0.01**	0.05	1.65
seq2.c	inc	to	to	nr	to	inc	0.98
up.c	nr	to	nr	**0.34**	to	1.25	0.54
nest-if3.c	nr	inc	to	nr	**0.21**	0.64	0.17
apache-get-tag.c	nr	to	to	**7.88**	to	to	1.48
seq-z3.c	nr	to	to	**2.9**	to	to	1.22
fragtest_simple.c	0.2	1.22	**0.01**	0.52	0.02	0.12	0.83
rajamani_1.c	0.2	1.26	0.01	0.16	**0.01**	0.03	5.13
nest-if8.c	nr	47.64	to	nr	inc	**0.02**	0.35
sendmail-mime-fromqp.c	0.21	1.71	0.04	0.05	0.02	**0.02**	0.38
gulwani_cegar2.c	0.2	14.62	nr	0.2	**0.03**	0.95	0.26
test.c	0.2	1.3	0.02	0.01	0.01	**0.01**	0.01
nested7.c	nr	inc	to	nr	inc	inc	0.98
id_build.c	nr	to	to	nr	**0.11**	0.3	0.42
svd1.c	nr	**3.22**	to	nr	inc	5.65	4.35
id_trans.c	0.43	18.43	29.26	0.69	**0.01**	0.02	0.55
nested5.c	nr	48.22	to	nr	**0.08**	to	0.23
gulwani_cegar1.c	nr	inc	0.03	0.21	**0.02**	0.04	0.43
ken-imp.c	**0.19**	1.66	nr	0.55	to	0.68	0.42
sort_instrumented.c	0.33	1.26	0.01	0.01	0.01	**0.01**	0.73
SpamAssassin-loop_ok.c	0.61	47.11	to	nr	**0.02**	0.08	0.7
up-nd.c	inc	to	26.74	1.87	**0.07**	1.17	0.95
compact_false.c	to	to	nr	to	to	**21.47**	0.29
veris.c_OpenSER__cases1_stripFullBoth_arr_true.c	0.2	19.69	to	nr	**0.02**	0.05	0.99
terminator_02_true.c	nr	1.59	1.34	0.16	**0.01**	0.03	0.38
bubble_sort_false.c	nr	**6.89**	to	err	err	err	1.96
n.c11_true.c	nr	1.28	nr	nr	**0.02**	inc	0.2

T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl

nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*

continued on next page

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
ludcmp_false.c	0.22	to	err	err	**0.1**	err	0.12
trex02_false.c	nr	1.36	0.39	0.16	0.02	**0.01**	0.37
matrix_true.c	nr	7.48	**0.04**	nr	err	err	0.62
vogal_false.c	nr	**1.61**	11.03	to	err	31.52	1.44
trex03_false.c	nr	1.7	8.13	0.06	**0.01**	0.02	0.85
while_infinite_loop_2_true.c	0.2	1.29	nr	0.03	**0.01**	0.02	0.2
insertion_sort_true.c	nr	to	to	nr	to	inc	0.51
matrix_false.c	nr	**22.86**	to	to	err	err	0.74
verisec_OpenSER__cases1_stripFullBoth_arr_false.c	nr	20.02	to	8.78	**0.17**	err	1.07
sum01_bug02_false.c	0.72	1.71	**0.15**	0.29	0.21	6.9	0.83
n.c24_true.c	nr	to	nr	to	to	to	0.6
veris.c_NetBSD-libc__loop_true.c	0.2	1.67	6.48	0.05	0.02	**0.01**	0.26
insertion_sort_false.c	nr	to	to	nr	to	**0.26**	0.63
sum01_false.c	0.77	1.99	**0.12**	0.15	0.38	0.72	0.46
for_infinite_loop_2_true.c	0.2	19.07	nr	nr	**0.01**	0.02	0.29
terminator_02_false.c	nr	1.4	3.25	0.2	**0.01**	0.01	0.42
nec20_false.c	nr	1.39	0.13	0.16	**0.02**	0.12	0.48
verisec_sendmail__tTflag_arr_one_loop_false.c	nr	2.09	to	nr	**0.78**	err	0.39
trex02_true.c	0.2	1.56	nr	nr	**0.01**	0.03	0.37
invert_string_false.c	nr	to	to	**3.65**	err	err	0.6
nec11_false.c	nr	1.56	0.14	0.07	**0.01**	0.02	0.23
bubble_sort_true.c	0.21	19.83	to	to	**0.02**	0.05	3.46
nec40_true.c	0.21	17.24	0.04	0.07	0.02	**0.01**	0.2
veris.c_sendmail__tTflag_arr_one_loop_true.c	0.2	2.19	0.24	nr	**0.01**	0.02	0.44
linear_sea.ch_true.c	nr	nr	nr	**0.18**	to	err	0.32
verisec_NetBSD-libc__loop_false.c	0.43	1.45	4.95	**0.04**	inc	err	0.25
sum01_true.c	nr	to	nr	**0.26**	to	1.24	0.5
sum04_true.c	0.2	1.24	**0.01**	0.08	0.03	0.28	0.31
count_up_down_false.c	0.22	1.41	0.14	0.05	**0.01**	0.02	0.24
trex03_true.c	nr	1.92	nr	nr	**0.03**	0.03	0.82

T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl

nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*

continued on next page

Benchmark	T_1	T_2	T_3	T_4	T_5	T_6	*Accl*
for_bounded_loop1_false.c	nr	1.44	16.87	0.8	**0.02**	0.09	0.67
n.c40_true.c	0.19	18.01	0.03	0.07	**0.01**	0.02	0.23
heavy_true.c	to	to	to	to	to	to	0.39
lu.cmp_true.c	0.48	1.45	**0.07**	err	24.42	err	0.19
terminator_01_false.c	nr	1.46	0.22	0.11	**0.01**	0.01	0.7
sum03_true.c	0.19	46.46	nr	0.15	**0.06**	0.22	0.38
linear_search_false.c	nr	nr	**0.25**	0.25	1.87	err	0.34
sum04_false.c	0.28	1.46	**0.03**	0.16	0.07	0.71	0.37
eureka_01_false.c	nr	to	to	to	1.34	**0.43**	1.75
sum01_bug02_sum01_bug02_base.case_false.c	0.54	1.7	0.15	0.63	0.12	**0.09**	0.54
while_infinite_loop_4_false.c	0.2	1.43	0.91	0.06	0.01	**0.01**	0.14
eureka_01_true.c	nr	**1.56**	15.39	to	to	7.31	1.15
while_infinite_loop_3_true.c	0.19	1.25	nr	nr	0.01	**0.01**	0.13
vogal_true.c	nr	to	**4.09**	4.19	54.34	to	1.63
count_up_down_true.c	**0.2**	to	nr	nr	to	to	0.27
while_infinite_loop_1_true.c	0.2	1.25	nr	0.03	**0.01**	0.01	0.21
for_infinite_loop_1_true.c	0.2	19.18	nr	nr	**0.01**	0.02	0.22
sum03_false.c	nr	1.52	0.78	0.25	**0.22**	0.69	0.76
invert_string_true.c	nr	7.16	**0.07**	0.96	0.14	inc	1.05
eureka_05_true.c	nr	1.37	**0.08**	0.48	0.87	0.1	0.64

T_1: UFO; T_2: CPAchecker; T_3: CBMC; T_4: CBMC + Accl; T_5: IMPARA; T_6: IMPARA + Accl

nr: no result; err: error; to: timeout; inc: incorrect result; *values are in seconds*

The source-level transformation of programs enables integration with futher invariant generation techniques. As a future work, we plan to investigate the interplay between acceleration and invariant generation to minimize the number of wrong alarms and to handle more cases correctly, including those that involve arrays. We also believe it would be worthwhile to investigate whether the accelerator can be assisted with additional invariants generated using some other technique (e.g. [20, 96]). Our initial experiments suggest that some of these invariants, even over the interval domain, may help us rule out the possibility of overflows, thereby increasing the precision of the accelerator.

3.5.1 Notes

The ability to accelerate invariant generation for loops enables one to tackle a complex system more effectively. However, in order to scale for real-world examples, it is important to address another, equally important, aspect of complex systems - the number of interacting components in them. There are several ways in which the components may interact: message passing, generating and sensing events, reading and writing to shared variables, etc. Moreover, the components may themselves be defined, or identified, in various ways. We address some of these concerns in the next chapter, as we develop techniques that can work compositionally.

Chapter 4

Exploiting Modularity of Implementation: Refutations

In software design, modularity refers to a logical partitioning of a system that allows complex software to be manageable for the purpose of implementation and maintenance. Given a design, or an implementation, this logical separation may either be apparent, e.g. components in a Statechart design, threads in a multi-threaded program, procedures in a sequential program, or it might need to be constructed explicitly, e.g. splitting a control-flow graph into sets of paths. In order to address scalability limitations of verification stemming from the *scale* of a system, it helps to analyze the system's modules in isolation and compose the results in the end. This chapter studies one such approach for refuting safety properties in sequential programs.

4.1 Compositional Safety Refutation

Divide-and-conquer approaches are considered to be the blue print solution to scale algorithms to large problems. Compositionality of proofs is the enabler of a map-reduce approach to verification. Compositional verification approaches based on contracts and summaries have been shown to tremendously increase scalability and productivity in real-world formal verification [6, 47, 77, 101].

But what about refutation? Unlike verification, refutation algorithms are usually based on finding a violating execution trace, which seems to be inherently non-compositional. Consequently, the study of the compositional refutation problem is an under-explored area of research. Yet, solutions to this problem have significant impact on other research problems. As a motivation, we give here two algorithmic approaches in verification and testing that will be enabled by efficient compositional refutation algorithms:

- Property-guided abstraction refinement algorithms like CEGAR [32] need to decide whether counterexamples that are found in the abstraction are spurious or true counterexamples. The lack of compositional refutation techniques forces these algorithms to operate in a monolithic manner and is therefore an obstacle to scaling them to large programs.
- Automated test generation techniques based on Bounded Model Checking are successfully used in various industries to generate unit tests (e.g. [97]). However, they do not sufficiently scale to accomplish the task of generating integration tests. Compositional refutation techniques achieve exactly this goal: they efficiently produce refutations (from which test vectors can be derived) on unit (module) level and enable their composition in order to obtain system level refutations, i.e. integration tests.

This chapter is a first step in this direction and lays the base for a more systematic study of the problem domain.

4.1.1 Contributions

We summarise the contributions of this chapter as follows.

- In order to place the problem in a wider context, we give an informal overview on how completeness relates to problem decomposition in safety refutation and verification (Section 4.3).
- We formalise the safety refutation problem in *horizontal decompositions*, e.g. procedure-modular decompositions, and characterise the compositional completeness guarantees of various algorithmic approaches (Section 4.4).
- We describe three refutation approaches with different degrees of completeness (Section 4.5) and give experimental results on C benchmarks, comparing their completeness and efficiency (Section 4.6).

4.2 Preliminaries

Before we introduce the notation, let us understand the goal that we have. Consider the program shown in Fig. 4.1. It has an assertion in the procedure *bar*. We are interested in finding a path in the program that reaches the assertion, with values of the variables such that the assertion does not hold. In particular, for this example, we wish to find a path starting from *main*, and reaching the assertion in *bar* with

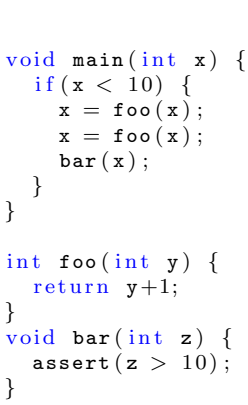
values of $z \leq 10$. Note that the assertion may be violated "locally" in *bar*, but we are interested in violations that are global, i.e. starting from the entry (*main*) function.

Program model and notation. We assume that programs are given in terms of acyclic[1] call graphs, where individual procedures f are given in terms of *deterministic*, symbolic input/output transition systems. F is the set of all procedures in the program. Since the handling of loops is orthogonal to the compositional aspect, we consider only loop-free procedures (respectively bounded unwindings of loops) in this chapter[2]. Thus, we simply denote the input/output relation of a procedure f as $T_f(\mathbf{x}^{in}, \mathbf{x}^{out})$. Inputs $\mathbf{x}^{in}$ are procedure parameters, global variables, and memory objects that are read by f. Outputs $\mathbf{x}^{out}$ are return values, and potential side effects such as global variables and memory objects written by f. Boolean *guard variables* (g) in inputs and outputs are used to model the control flow. Non-deterministic conditionals and variable initializations are modeled by a call to a *nondet()* function which returns a non-determistic value. During analysis this return value is treated like an additional input, in order to make the transition system deterministic. The relations T_f are given *as first-order logic formulae* over bitvectors and arrays, resulting from the logical encoding of the program semantics. Fig. 4.1 gives an example of the encoding of a program into such formulae using the loop-free notation. The inputs $\mathbf{x}^{in}$ of *foo* are (y, g_6) and the outputs $\mathbf{x}^{out}$ consist of (r, g_7) where r is the return value. In addition to the inputs and outputs we need boolean guard variables g^{in}, g^{out} (here g_6, g_7) that are set to true if the entry (exit) of the procedure is backward (forward) reachable. They are handled like input/output parameters and have their actual counterparts in the guard variables in the caller (here, e.g. g_1, g_2 for the call foo_0 in *main*). Note that we consider exit in a procedure is not reachable, i.e., $\neg g^{out}$, if either the program is non-terminating or an assertion in a procedure is violated. Hence, the exit guard condition in the definition of a transition function includes assertion checks as in T_{bar}. We use a single static assignment (SSA) encoding, which gives a fresh name to each update of a variable if it is modified multiple times, such as for example in *main*.

Each call to a procedure h at call site i in a procedure f is modeled by a *placeholder predicate* $h_i(\mathbf{x}^{p\_in}{}_i, \mathbf{x}^{p\_out}{}_i)$ occurring in the formula T_f for f. The placeholder predicate ranges over intermediate variables in the SSA of caller, representing its actual input and output parameters, $\mathbf{x}^{p\_in}{}_i$ and $\mathbf{x}^{p\_out}{}_i$, respectively. Placeholder predicates evaluate to *true* in the beginning, which corresponds to havocing the program variables in

[1] We consider non-recursive programs with multiple procedures (cf. model in [28]).

[2] Section 4.7 discusses the extension to programs with loops.



```
void main(int x) {
  if(x < 10) {
    x = foo(x);
    x = foo(x);
    bar(x);
  }
}

int foo(int y) {
  return y+1;
}
void bar(int z) {
  assert(z > 10);
}
```

$$
\begin{aligned}
T_{main}((x_0, g_0), (g_5)) &\equiv g_1{=}(g_0 \wedge (x_0{<}10)) \wedge \\
&\quad foo_0((x_0, g_1), (x_1, g_2)) \wedge \\
&\quad foo_1((x_1, g_2), (x_2, g_3)) \wedge \\
&\quad bar((x_2, g_3), (g_4)) \wedge \\
&\quad g_5{=}(g_0 \wedge \neg(x_0{<}10) \vee g_4) \\
Props_{main} &\equiv true \\
T_{foo}((y, g_6), (r, g_7)) &\equiv (r{=}y{+}1) \wedge (g_6{=}g_7) \\
Props_{foo} &\equiv true \\
T_{bar}((z, g_8), (g_9)) &\equiv g_9{=}(g_8 \wedge (z{>}10)) \\
Props_{bar} &\equiv g_8 \Rightarrow (z{>}10)
\end{aligned}
$$

Figure 4.1: Example program and its encoding

procedure calls. As the analysis progresses, they get strengthened by summaries. We later explain how we use the guard variables in performing this propagation. In procedure *main* in Fig. 4.1, the placeholder for the first procedure call to *foo* is $foo_0((x_0, g_1), (x_1, g_2))$ with the actual input and output parameters x_0, x_1, respectively, and the corresponding guard variables that encode whether the entry and exit of foo_0 are reachable. Let $Props_f$ denote the conjunction of all properties (assertions) in procedure f (e.g. the assertion in *bar* in Fig. 4.1). Note that we view these relations as predicates, e.g. $T(\mathbf{x}, \mathbf{x}')$, with given parameters $\mathbf{x}, \mathbf{x}'$, and mean the $T[\mathbf{a}/\mathbf{x}, \mathbf{b}/\mathbf{x}']$ when we write $T(\mathbf{a}, \mathbf{b})$. Moreover, we write $\mathbf{x}$ and x with the understanding that the former is a vector, whereas the latter is a scalar.

CS_f is the set of call sites in procedure f, and the set of all call sites in a program, CS, is $\bigcup_{f \in F} CS_f$. $func(i)$ is the procedure called at call site i. We write X_f for the variables in T_f (including intermediate variables), and $\hat{X}$ for the entirety of variables in $T_{func(i)}(\mathbf{x}^{in}{}_i, \mathbf{x}^{out}{}_i)$ for all $i \in CS$.

Summaries, and Calling Contexts Inter-procedural compositional proofs of a sequential program usually use a set of auxiliary predicates to define abstractions of loops and procedures. These abstractions are usually formally defined by means of a set of predicates – *invariants*, a *summary* and a *calling context* ($CallCtx_{func(i)}$) for every procedure invocation at call site i in a call-graph of the program. These predicates have the following roles: Invariants abstract the behaviour of loops inside functions. Summaries abstract the behaviour of called procedures; they are used to strengthen the placeholder predicates. Calling contexts abstract the caller's behaviour w.r.t. the procedure being called. When analyzing the callee, the calling contexts are used to constrain its inputs and outputs. The set of sub-traces (in the behaviour

of a caller), corresponding to execution of a function at a call site, is characterised by a conjunction of the calling context and summary predicates associated with the function at that call site. We provide formal definitions for summaries and calling contexts below (invariants are not needed in this chapter, except for Section 4.7 where we discuss the extension to programs with loops).

Definition 4.2.1. *For a procedure given by T_f we define:*

- *A* summary *is a predicate Sum_f such that:*

$$\forall X_f : \quad T_f(\mathbf{x}^{in}, \mathbf{x}^{out}) \Longrightarrow Sum_f(\mathbf{x}^{in}, \mathbf{x}^{out})$$

- *The* calling context *for a procedure call at call site i in the given procedure is a predicate $CallCtx_{func(i)}$ such that*

$$\forall X_f : \quad T_f(\mathbf{x}^{in}, \mathbf{x}^{out}) \Longrightarrow CallCtx_{func(i)}(\mathbf{x}^{p\_in}{}_i, \mathbf{x}^{p\_out}{}_i)$$

For instance, a summary for procedure *foo* in Fig. 4.1, is $Sum_{foo}((y, g_6), (r, g_7)) = (y{<}MAX \Rightarrow r{>}y)$.[3] A (forward) calling context for the first call to procedure *foo* in *main* is $CallCtx_{foo_0}((x_0, g_1), (x_1, g_2)) = (g_1 \Rightarrow x_0{<}0)$. We observe that the guard variables are also used in defining summaries and calling contexts. They have the same meaning as in transition functions. The reason we have defined *CallCtx* over both input and output parameters is so we can propagate it in forward or backward directions. With a slight abuse of notation, we sometimes use only the call site index or the function name, if there's no ambiguity, as a subscript to specify *CallCtx*.

4.3 Compositional Verification and Refutation Overview

A decomposition of a verification problem intuitively splits the original problem into a set of sub-problems that cover the original problem. The decomposition operator for the problem has a corresponding composition operator for composing the results obtained from the sub-problems in order to obtain a solution of the original problem. Compositionality has been naturally studied in the context of the parallel composition of processes (e.g. [33, 35, 89]) where the decomposition is performed according to the process structure and the composition operator is a rely-guarantee proof rule, for example.

[3] MAX denotes the maximum possible value in the type of y.

In terms of program executions, a decomposition can be viewed as a way a proof of verification splits the behaviour, i.e. the set of all execution traces of a program, in constructing the proof. For sequential programs, decompositions can be *vertical* or *horizontal*. Decomposition of a verification problem can be formally defined by means of a pair of operators – *decompose* that decomposes a program into *modules* and a *compose* operator that composes proof for a module in terms of its sub-modules.

A *vertical* decomposition usually focuses on entire execution traces and splits the behaviour of the program into subsets of end-to-end traces. Program slicing (e.g. [62]) that splits verification into a set of use-case scenarios and then using symbolic execution for checking each component is an example of a vertical decomposition. An automata-based semantic decomposition of programs was proposed by [64]. The decompose-compose pair of operators for such a vertical decomposition can be defined as follows.

Definition 4.3.1 (Vertical decomposition-composition)**.** *A vertical decomposition of a program, Prog, is defined as*

$$decompose(Prog) =_{def} P,$$

where P is a collection of (sets of) program behaviors, i.e. execution paths of the program. A safety proof of Prog may be obtained with the help of a composition operator, compose, that combines the individual proofs of every element $p \in P$, i.e.

$$proof(Prog) = compose(p \in P,\ proof(p))$$

.

A *horizontal* decomposition is usually based on a syntactic decomposition of the program e.g. into procedures or *procedures*. In terms of traces, horizontal decomposition splits execution traces into pieces, i.e. each element of the decomposed program captures a set of sub-traces corresponding to each procedure invocation in the program.

We now define the decomposition and composition operators w.r.t. a horizontal decomposition based on the procedure call hierarchy in a program.

Definition 4.3.2 (Horizontal decomposition-composition)**.** *A horizontal decomposition of a program Prog is defined as*

$$decompose(Prog) =_{def} M,$$

where M is a collection of procedures in the call graph of Prog.

Algorithm 2 Composition operator for summaries

```
1: procedure COMPOSE(f)
2:     for all i ∈ CS_f do                          ▷ CS_f are the call sites in procedure f
3:         Sum_{func(i)} ← COMPOSE(func(i))         ▷ func(i) is the procedure at call site i
4:     Sum_f ← proof(f)      ▷ uses Sum_{func(i)}, i ∈ CS_f and proof composer operator
5:     return Sum_f                                 ▷ Sum_f can be cached
```

Let $CS_f(f)$ denote the set of call sites in a procedure f and $func(i)$ denote the procedure called at site i.

The safety proof for a procedure f in a horizontal decomposition is defined with the help of a composition operator as

$$proof(f) =_{def} compose(i \in CS_f,\ func(i))$$

.

The proof of Prog may be obtained as $proof(f_{entry})$, where f_{entry} is the entry procedure of Prog.

Consider a safe version of the code in Fig. 4.1 where the assertion in *bar* is changed to $z \leq 10$. A safety proof for the program can be constructed hierarchically by using the following summaries for *foo* and *bar*: $Sum_{foo}((y, g_6), (r, g_7)) = (r{=}y + 1 \wedge g_6{=}g_7)$ and $Sum_{bar}((z, g_8), (g_9)) = (g_9 \Rightarrow z \leq 10)$. Then, the proof for *main* can be constructed using the recursive Algorithm 2. The proof for the leaves (*foo* and *bar*) involves showing their transition functions imply their respective summary. Proof composition for a non-leaf procedure will use the caller summaries to similarly construct a proof (a summary) for the caller. In our example, the program is indeed proved safe as the algorithm constructs a Sum_{main}, which, in this case, can be a suitable abstraction of the transition function for *main*, that is not *false*, while checking that the constructed summaries verify all the embedded properties.

This chapter focuses on solving the refutation problem with horizontal decompositions.

The challenge in automating horizontal compositional verification lies in synthesising a set of precise summary predicates for the procedures in the call graph. Note that in the program in Fig. 4.1, it was essential to constrain the input z to *bar* as $(z \leq 1)$ to get a proof. This effort is made harder if the code has loops, which require *invariants* and use of abstractions of loops and procedures. The calling contexts and summaries can be mutually dependent even for non-recursive programs. In general, one requires iterative fix-point computation on the call-graph structure, possibly using

abstraction and refinement. A pre-requisite for performing abstraction refinement is the ability to refute safety and check for spurious counterexamples also in a modular and efficient fashion, which is the goal of this chapter.

A Practical View of the Modular Refutation Problem. Consider the example in Fig. 4.1 in Section 4.2. This program is unsafe because when *bar* is called the actual argument to it that takes the place of z can at most be only 1. The question is if we can arrive at this refutation modularly. Analysing procedure *bar* in isolation indeed gives a counterexample, which could be possibly spurious.

Instantiated on the example in Fig. 4.1, a refutation involves checking $\neg\forall z, g_8 : g_8 \Rightarrow (z > 10)$. A counterexample could be $g_8 \wedge z = 5$, for example. The question is now how to decide whether this counterexample is spurious or not, and to find a valid counterexample if one exists. For instance, $z = 5$ turns out to be spurious if we consider the whole program because it clashes with $x_0 < 0$ in *main*. However, $z = -8$ would be a valid counterexample.

The set of *local* counterexamples found in a procedure f might contain many counterexamples that are spurious for the whole program, i.e. they are infeasible from the entry point of the program. That is why a definite answer to this problem of finding a refutation modularly, if one exists, cannot be given by only analyzing the procedures in isolation. This is the reason why refutation in horizontal decompositions is hard — unlike refutation in vertical decompositions where a refutation of the local problem implies the refutation of the global one.

Intuitively, the negation of the *assertion* has to be hoisted up along the error path to the entry point of the program. If the obtained weakest precondition for the violation of the assertion is not *false*, then the counterexample is feasible. Propagating up the *counterexample* itself is not sufficient to decide spuriousness as illustrated above.

4.4 Formalising Horizontal Compositional Refutation

In this section we formalise the problem of safety refutation for sequential programs. To simplify the presentation we focus on *loop-free* programs. The formalisation for programs with loops is structurally similar, but in addition, requires the handling of invariants (see Section 4.7), which is orthogonal to the compositional aspect.

We give three different formalisations – the first corresponds to a monolithic approach, and the remaining two correspond to compositional approaches.

4.4.1 Monolithic Safety Refutation Problem

For non-recursive programs, since one can always inline every procedure call at its call site, we can replace every call by recursively inlining its body. Then, to *refute* safety we have to show invalidity of the following formula:

$$\forall \hat{X} : \bigwedge_{j \in CS} g^{in}_{f_{entry}} \wedge T_{func(j)}(\mathbf{x}^{in}{}_j, \mathbf{x}^{out}{}_j) \wedge InlineSums_{func(j)} \Rightarrow Props_{func(j)}(\mathbf{x}_j) \tag{4.1}$$

where

- $InlineSums_f$ is $\bigwedge_{i \in CS_f} InlineSum_{func(i)}(\mathbf{x}^{p\_in}{}_i, \mathbf{x}^{p\_out}{}_i)$,
- $InlineSum_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f)$ is $T_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge InlineSums_f$,
- $Props_f(\mathbf{x}_f)$ is the conjunction of all properties (assertions) in the procedure f,
- $\hat{X}$ is the entirety of variables in (4.1),
- and the conjunction with $g^{in}_{f_{entry}}$ states that the entry procedure is reachable.[4]

In the formula above, we have included the summary of all call sites. It is noteworthy that if the call sites fall under branches in the program that are mutually exclusive (e.g. if(*) then *foo* else *bar*), not all of them would get exercised at once in any execution. However, there is no explicit notion of execution in a formula. In our notation, this is handled by adding the branching condition to the entry guards for such call sites. Thus, the summary of all call sites will lie there in the formula, without affecting the semantics of the execution.

Alternatively, we can write:

$$\begin{aligned} \exists \overbrace{Sum_f, \ldots :}^{\text{for all } f \in F} \bigwedge_{f \in F} \forall X_f : \\ \big(g^{in}_{f_{entry}} \wedge T_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge Sums_f \implies Props_f(\mathbf{x}_f)\big) \\ \wedge \quad \big(T_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge Sums_f \iff Sum_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f)\big) \end{aligned} \tag{4.2}$$

where $Sums_f$ is $\bigwedge_{i \in CS_f} Sum_{func(i)}(\mathbf{x}^{p\_in}{}_i, \mathbf{x}^{p\_out}{}_i)$.

This formulation uses a predicate Sum_f to exactly express the behaviours of each procedure f. 4.1 is valid iff 4.2 is valid. The existential quantifier in (4.2) can be shown to be uniquely eliminated by recursively replacing the Sum_f predicates by left-hand side of the equivalence in the last line in (4.2), obtaining (4.1). Note that solving (4.1) is NP-complete, whereas solving (4.2) is PSPACE-complete [8]. However, (4.1) may be exponentially larger (in the number of variables) than (4.2).

[4]This amounts to using $T_{f_{entry}}[true/g^{in}_{f_{entry}}]$ as the transition relation of f_{entry}.

Both versions are *monolithic* because they consider the entire program as a whole. In particular, (4.2) finds summaries *globally*, i.e. for the whole program.

Also note that, proving invalidity of (4.2) shows the inexistence of a verification proof, but it does not directly allow us to derive a counterexample in terms of an execution trace because of the universal quantification of the variables. Moreover, showing unsatisfiability of (4.2) is difficult because it involves proving the inexistence of summary predicates. For this reason, many practical techniques, such as SAT-based Bounded Model Checking use (4.1) (considering bounded unwindings for programs with loops in order to make them loop-free). Note that negating (4.1) results in an existentially quantified problem, whose satisfiability witnesses a refutation in the form of values for the variables $\hat{X}$.

However, solving (4.1) monolithically is often intractable. Therefore, we want to decompose the problem into smaller subformulae that are faster to solve. (4.2) is amenable to decomposition, but it does not allow us to approximate the summaries with the help of abstractions (because of $\Leftrightarrow$ in last line). Therefore we give a third formulation of the monolithic problem that additionally uses *calling contexts*. The calling context for the entry procedure is $g^{in}_{f_{entry}}$.

$$
\begin{aligned}
&\exists \overbrace{Sum_f, CallCtx_f, \ldots :}^{\text{for all } f \in F} \textstyle\bigwedge_{f\in F} \forall X_f : \\
&\quad \big(CallCtx_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge \\
&\quad T_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge Sums_f \implies \quad Props_f(\mathbf{x}_f) \wedge \\
&\qquad\qquad\qquad\qquad\quad Sum_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge \\
&\qquad\qquad\qquad\qquad\quad \textstyle\bigwedge_{j\in CS_f} CallCtx_{func(j)}(\mathbf{x}^{p\_in}{}_j, \mathbf{x}^{p\_out}{}_j)\big)
\end{aligned}
\tag{4.3}
$$

Eq. (4.3) is equisatisfiable with (4.2), although (4.3) admits more solutions to Sum_f including those that are over-approximations adequate to prove the properties. To see this, if (4.2) is satisfiable, the precise solution of (4.2) for Sum_f can be used to satisfy (4.3) by plugging it in for both $CallCtx_f$ and Sum_f in (4.3). If (4.2) is unsatisfiable, then so is (4.3) because one or all behaviours included in Sum_f solution of (4.2) violates one of the properties. Then, every solution to (4.3) would violate the properties as they are over-approximations of the precise summaries. Note that this formulation is still monolithic because it requires one to synthesize all the contexts and summaries simultaneously by solving (4.3) corresponding to the entire program.

4.4.2 Modular Safety Refutation Problem

Let us now have a look at the *horizontal* decomposition following the procedural structure of the program. The goal is to compute the summaries Sum_f for each f while

considering only f and the summaries for the procedures called in f. We can attempt at achieving this by flipping the existential quantifier ($\exists Sum_f$) and the top-level conjunction ($\bigwedge_{f \in F}$ in (4.3)). However, this does not result in an equisatisfiable formula because existential quantification does not distribute over conjunctions. Therefore, we need an alternative formulation to solve the existential query per procedure. One approach is to search for a minimal solution for summaries and calling contexts occurring within each calling site of procedure f for a given context for f that satisfies all the embedded properties in f as shown in 4.4. I.e. for each $f \in F$ we have:

$$\begin{aligned} \min Sum_f, \overbrace{CallCtx_j, \ldots}^{\text{for all } j \in CS_f} : \forall X_f : \\ \big(CallCtx_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge \\ T_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge Sums_f \implies \quad & Props_f(\mathbf{x}_f) \wedge \\ & Sum_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge \\ & \textstyle\bigwedge_{j \in CS_f} CallCtx_j(\mathbf{x}^{p\_in}{}_j, \mathbf{x}^{p\_out}{}_j)\big) \end{aligned} \tag{4.4}$$

The operator $\min P : F(P)$, used in equation 4.4 above, is defined w.r.t. implication order for a formula F involving predicates P, i.e. as $\exists P : F(P) \wedge \forall P' : (P' \Rightarrow P) \Rightarrow \neg F(P')$. Note that $\min P$ is not unique in a partial order on predicates. In (4.4), min distributes over the conjunction of all $\forall$ formulae. In other words, for each $f \in F$, it gives a solution for Sum_f and the calling contexts for all embedded calling sites relative to a $CallCtx_f$, assuming there is a minimal solution for summaries for all the embedded procedures. But, we have not broken the dependency between calling contexts and summaries. Solving this problem requires computing a fixed point in the composition operator (presented below) and computing minimal solutions for the summary and calling context predicates. That is, what has been an existential second-order satisfaction problem in (4.3), has now become a second-order minimisation ($\exists\forall$) problem. The reason for this is that the mere existence of a solution for Sum_f and $CallCtx_{func(j)}$ does not prove that the overall verification problem holds. Therefore, we pessimistically have to assume that we require the *exact* calling contexts and summaries in order to *decide* the problem during proof composition.

The proposed proof composition operator (*compose*) with calling contexts is shown in Alg. 3 and is more complex than Alg. 2. The term "Solve" on line 5 refers to computing minimal summaries and calling contexts. The idea is to use the call graph of the program to compute the minimal calling context for each call site of procedure call of f piecewise in a top-down fashion, and use that calling context to compute a piecewise minimal summary for f for that call site (note the conjunction on Line 12 of Alg. 3) consistent with all the properties in f. The piecewise summaries and contexts

Algorithm 3 Composition operator with calling contexts

1: **global** $Sum_f \leftarrow \neg g_f^{out}$ for all $f \in F$
2: $CallCtx_f \leftarrow false$ for all $f \in F$
3: **procedure** $compose(f,\ CallCtx_f^*)$
4: **while** *true* **do** ▷ Repeat until fixed point reached
5: Solve (4.4) with $CallCtx_f^*$ as $CallCtx_f$ ▷ obtain Sum_f & $CallCtx_j$ for all $j \in CS_f$
6: **for all** $j \in CS_f$ **do** ▷ join calling contexts for $func(j)$
7: $CallCtx_{func(j)} \leftarrow CallCtx_{func(j)} \vee CallCtx_j(\mathbf{x}^{in}{}_{func(j)}, \mathbf{x}^{out}{}_{func(j)})$
8: **if** $CallCtx_{func(j)}$ for all $j \in CS_f$ has not changed **then**
9: **return** Sum_f
10: **for all** $j \in CS_f$ for which $CallCtx_{func(j)}$ has changed **do**
11: $Sum_j \leftarrow compose(func(j), CallCtx_j(\mathbf{x}^{in}{}_{func(j)}, \mathbf{x}^{out}{}_{func(j)}))$
12: $Sum_{func(j)} \leftarrow Sum_{func(j)} \vee (CallCtx_j(\mathbf{x}^{in}{}_{func(j)}, \mathbf{x}^{out}{}_{func(j)}) \wedge Sum_j)$
13: ▷ join summaries for $func(j)$

are combined disjunctively as they are built, which takes care of the dependency between summary and calling contexts. In the algorithm, each time *compose* is called recursively for f, it is called with a new piece of entry calling context for f and (4.4) is solved with summaries computed up to that point for the procedures in the body of f. Solving the equation may result in new contexts for each call site (if any) inside f and a new piece of summary for f all of which are accumulated.

For a program with entry function f_{entry}, a proof can be constructed by calling $compose(f_{entry}, g^{in}_{f_{entry}})$. The calling context $g^{in}_{f_{entry}}$ means that the entry procedure is reachable. The calling context of all embedded functions are initialised to *false* as that is the least element and also makes everything following the first call site unreachable. The summary for each f is initialised to $\neg g_f^{out}$, meaning that its exit is not reachable and hence execution cannot continue beyond any call to f. This initial value for summary has the effect of blocking analysis of all functions following f in the code until a piecewise summary is computed for f.

Observe that, as opposed to monolithic (4.3) where the fixed point computation for resolving the mutually dependent summary and calling context predicates (cf. [95]) is done within the solver for solving the monolithic formula, the fixed point in the modular version must be computed during the composition of the individual results. I.e. we have to saturate the Sum_f and $CallCtx_f$ predicates.

Theorem 4.4.1. *We obtain $Sum_{f_{entry}} = false$ using Alg. 3 iff (4.3) is unsatisfiable. I.e. horizontal decomposition is sound and complete.*

Proof (sketch): ($\Rightarrow$) The forward direction is easy. Alg. 3 will return a *false* summary only if Eq. (4.4) fails to give a minimal solution for summaries and calling contexts. Since (4.4) is simply a transformation of the existential second-order satisfaction problem of (4.3) into a second-order minimisation problem, it follows that if no minimal solutions exist (for 4.4), then (4.3) is also unsatisfiable.

($\Leftarrow$) We argue by induction on the depth (k) of the top-level procedure in the call graph of the program.

For the base case ($k = 1$), there is only one procedure call - the call to the entry function, f_{entry}. Since the calling context of f_{entry} is $g^{in}_{f_{entry}}$, and there are no other procedure calls, it is evident that computing $Sum_{f_{entry}}$ from Alg. 3 effectively reduces to finding it by solving (4.4) (line 5 of Alg. 3), with *Sums* and $CallCtx_j$ not present. This makes Eq. (4.4) and (4.3) identical and hence the statement follows trivially.

Assuming that the statement holds for all procedures in the call graph with depth $\leq k$, we will argue that it also holds if the entry function is at depth k.

Assume (4.3) is unsatisfiable. Then there must exist a function h which appears somewhere in the call graph, such that i) when *compose* is called for h, it returns $Sum_h = false$ (by induction hypothesis, because (4.3) is unsatisfiable for h), or ii) Sum_h conflicts with T_f. In both these cases, Alg. 3 simply returns *false*. In the first case, it is due to one of the embedded summaries becoming *false*, while in the second case it is just due to the contradiction arising at the current level.

4.4.3 Modular Safety Refutation with Witnesses

(4.4) suffers from the same problem as (4.3) that we cannot extract counterexamples in terms of an execution trace in case of a refutation because the formulae are unsatisfiable for refutations (i.e. Alg. 3 just returns false), and thus the solver does not return a countermodel. Therefore, we give next a formulation and a corresponding composition operator that produces refutation witnesses. The idea here is to compute piecewise contexts and summaries backwards starting from exit points of each procedure, much like a weakest-precondition computation works. Additionally, we start with negation of properties and compute maximal summary and contexts that possibly lead the program to an error state. In other words, a summary computed for f represent maximal symbolic witness to all the states reachable to safety violation. Such a summary can be obtained as maximal solutions to the equation shown in 4.5.

$$\begin{aligned}
&\max Sum_f, \overbrace{CallCtx_j, \ldots}^{\text{for all } j \in CS_f} : \forall X_f : \\
&\quad Sum_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge \\
&\quad \textstyle\bigwedge_{j \in CS_f} CallCtx_j(\mathbf{x}^{p_in}{}_j, \mathbf{x}^{p_out}{}_j) \implies (CallCtx_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \vee \neg Props_f) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad T_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge Sums_f
\end{aligned} \tag{4.5}$$

where $\max P.F(P)$ is defined as usual: $\exists P.F(P) \wedge \forall P'.(P \Rightarrow P') \Rightarrow \neg F(P')$.

(4.5) describes maximal solutions for the summary and calling contexts that are *contained* in the behaviour of the procedure. That is the reason the predicates for the summary and the calling contexts (for the called functions) appear on the left-hand side of the implication and the transition relation is on the right-hand side, i.e. reversed in comparison with (4.4). The disjuncts in the first part of the consequent of (4.5) are the sources of safety violations: these are safety violations in the caller (which are propagated by $CallCtx_f$), and safety violations in f itself ($\neg Props_f$). Safety violations in callees are propagated through the summaries. Both these are constrained to be consistent with the transition relation of f (with current summaries plugged in for the called functions), which ensures spurious errors are not propagated upwards.

We use the composition operator as in Alg. 3, but with the following modifications to the initialization. We call this composition operator *compose′* or Alg. 3' from now on.

- Initially, $Sum_f \leftarrow \neg g_f^{in}$ for all $f \in F$, meaning that the entry of f is not backwards-reachable.
- In Line 5, we solve (4.5).

The calling contexts for all embedded functions are initialized to *false* as before except for the top-level function f_{entry}. A refutation is constructed by computing $compose'(f_{entry}, \neg g_{f_{entry}}^{out})$. The calling context $\neg g_{f_{entry}}^{out}$ of f_{entry} means that we cannot reach the regular exit of the entry procedure if there is a property violation. If there are no property violations at this level (or no properties), then this choice for top-level context would still work as the second conjunct in equation 4.5, which denotes the transition relation, would ensure the precise contexts propagated to the first embedded call site from exit point. The choice of initial summary of $\neg g_f^{in}$ for all embedded functions will ensure that the summaries are generated in order of dependency of function calls backward from the exit point.

Theorem 4.4.2. *We obtain $Sum_{f_{entry}}$ using Alg. 3' such that $\exists \mathbf{x}^{in}, \mathbf{x}^{out} : g_{f_{entry}}^{in} \wedge Sum_{f_{entry}}(\mathbf{x}^{in}, \mathbf{x}^{out})$ iff (4.3) is unsatisfiable.*

Note that the conjunction with $g^{in}_{f_{entry}}$ projects the summary on the inputs, which must be satisfiable to have a refutation.

Proof (sketch): From Theorem 4.4.1, the statement of the theorem above can be simplified as $Sum_{f_{entry}} = false$ *using Alg. 3 iff* $Sum_{f_{entry}} \wedge g^{in}_{f_{entry}}$ *is satisfiable using Alg. 3'*. In other words, $Sum_{f_{entry}} \wedge g^{in}_{f_{entry}}$ *is satisfiable using Alg. 3' iff there is a refutation.* Further, note that the summaries and calling contexts are computed in Alg. 3' such that their projection on the input variables of a procedure is the weakest precondition w.r.t. the negation of the property ($\neg Props$).

($\Rightarrow$) This direction is easy to see. As the summaries and calling contexts are extracted from the weakest precondition w.r.t. $\neg Props$, a satisfying assignment to $Sum_{f_{entry}}$ is nothing but a refutation witness. Since $Sum_{f_{entry}}$ is satisfiable, it immediately follows that the property is refutable.

($\Leftarrow$) We argue this by induction on the number (k) of procedure calls.

For the base case ($k = 1$), the only function being executed is the entry function, f_{entry}. Alg. 3', therefore, simply reduces to solving equation 4.5 and obtaining the summary from there. From the way we arrived at (4.5), it is clear that the $Sum_{f_{entry}}$ is satisfiable in case there is a refutation.

Suppose the statement holds for programs with $k \leq n$. Consider a program with $(n+1)$ procedure calls. If we look at the entry function of this program, there may be execution paths starting in f_{entry} having at most n procedure calls. From the induction hypothesis, it follows that if a refutation is possible along one of these paths, the error summary for the top-most function along that path is satisfiable. If we propagate the error summaries along each of these paths by computing the weakest precondition, we can claim that a refutation is possible only if these piecewise summaries, combined disjunctively, is satisfiable at the function entry. This is exactly what Alg. 3' captures. Thus, even for this program with $(n+1)$ calls, it follows that if the entry function is reachable, a refutation implies that $Sum_{f_{entry}}$ is satisfiable.

4.4.4 Worked Example

Let us consider the example in Fig. 4.1, but with the conditional in line 2 being $x < 10$. We start with $Sum_{main}((x_0, g_0), (g_5)) = \neg g_0$, $Sum_{foo}((y, g_6), (r, g_7)) = \neg g_6$, $Sum_{bar}((z, g_8), (g_9)) = \neg g_8$, and $CallCtx^*_{main}((x_0, g_0), (g_5)) = \neg g_5$, $CallCtx_{foo}((y, g_6), (r, g_7)) = false$, $CallCtx_{bar}((z, g_8), (g_9)) = false$.

The composition operator is called for *main*. We solve (4.5):

$$\begin{aligned}
&\max Sum_{main}, CallCtx_{foo_0}, CallCtx_{foo_1}, CallCtx_{bar} : \forall X_{main} : \\
&\quad Sum_{main}((x_0, g_0), (g_5)) \wedge \\
&\quad CallCtx_{foo_0}((x_0, g_1), (x_1, g_2)) \wedge \\
&\quad CallCtx_{foo_1}((x_1, g_2), (x_2, g_3)) \wedge \\
&\quad CallCtx_{bar}((x_2, g_3), (g_4)) \implies \quad (\neg g_5 \vee \neg true) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad g_1 = (g_0 \wedge (x_0 < 10)) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad g_5 = ((g_0 \wedge \neg(x_0 < 10)) \vee g_4) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad \neg g_1 \wedge \neg g_2 \wedge \neg g_3
\end{aligned}$$

We obtain the following solutions for the predicates: $CallCtx_{bar} = \neg g_4$, $CallCtx_{foo_1} = \neg g_3$, $CallCtx_{foo_0} = \neg g_2$, $Sum_{main} = \neg g_0 \wedge \neg g_5$.

Then we recur into *bar* with (4.5) instantiated as:

$$\begin{aligned}
&\max Sum_{bar} : \forall z, g_8, g_9 : \\
&\quad Sum_{bar}((z, g_8), (g_9)) \implies \quad (\neg g_9 \vee \neg(g_8 \Rightarrow z > 10)) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad (g_9 = (g_8 \wedge z > 10))
\end{aligned}$$

Hence, we get for Sum_{bar}: $(g_8 \Rightarrow \neg(z > 10)) \wedge \neg g_9$.

In Line 6 of Alg. 3', (4.5) for *main* is then:

$$\begin{aligned}
&\max Sum_{main}, CallCtx_{foo_0}, CallCtx_{foo_1}, CallCtx_{bar} : \forall X_{main} : \\
&\quad Sum_{main}((x_0, g_0), (g_5)) \wedge \\
&\quad CallCtx_{foo_0}((x_0, g_1), (x_1, g_2)) \wedge \\
&\quad CallCtx_{foo_1}((x_1, g_2), (x_2, g_3)) \wedge \\
&\quad CallCtx_{bar}((x_2, g_3), (g_4)) \implies \quad (\neg g_5 \vee \neg true) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad g_1 = (g_0 \wedge (x_0 < 10)) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad g_5 = ((g_0 \wedge \neg(x_0 < 10)) \vee g_4) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad \neg g_1 \wedge \neg g_2 \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad (g_3 \Rightarrow \neg(x_2 > 10)) \wedge \neg g_4
\end{aligned}$$

which results in $CallCtx_{bar} = \neg g_4$, $CallCtx_{foo_1} = g_3 \Rightarrow \neg(x_2 > 10)$, $CallCtx_{foo_0} = \neg g_2$, $Sum_{main} = \neg g_5$. Hence, $CallCtx_{foo}$ is updated to $g_7 \Rightarrow \neg(r > 10)$.

In the next iteration of *compose*(*main*) we recur into foo_1 and solve:

$$\begin{aligned}
&\max Sum_{foo} : \forall y, g_6, r, g_7 : \\
&\quad Sum_{foo}((y, g_6), (r, g_7)) \implies \quad ((g_7 \Rightarrow \neg(r > 10)) \vee \neg true) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad (g_6 = g_7) \wedge (r = y + 1)
\end{aligned}$$

Thus, Sum_{foo} is updated to $(g_6 \Rightarrow \neg(r > 10) \wedge g_7) \wedge (r = y + 1)$.

Then in Line 6 in $compose(main)$, we solve

$$\begin{aligned}
&\max Sum_{main}, CallCtx_{foo_0}, CallCtx_{foo_1}, CallCtx_{bar} : \forall X_{main} : \\
&\quad Sum_{main}((x_0, g_0), (g_5)) \wedge \\
&\quad CallCtx_{foo_0}((x_1, g_2)) \wedge \\
&\quad CallCtx_{foo_1}((x_2, g_3)) \wedge \\
&\quad CallCtx_{bar}((g_4)) \implies \quad (\neg g_5 \vee \neg true) \wedge \\
&\qquad\qquad\qquad\qquad\quad g_1 = (g_0 \wedge (x_0 < 10)) \wedge \\
&\qquad\qquad\qquad\qquad\quad g_5 = ((g_0 \wedge \neg(x_0 < 10)) \vee g_4) \wedge \\
&\qquad\qquad\qquad\qquad\quad (g_1 \Rightarrow \neg(x_1 > 10) \wedge g_2) \wedge (x_1 = x_0 + 1) \wedge \\
&\qquad\qquad\qquad\qquad\quad (g_2 \Rightarrow \neg(x_2 > 10) \wedge g_3) \wedge (x_2 = x_1 + 1) \wedge \\
&\qquad\qquad\qquad\qquad\quad (g_3 \Rightarrow \neg(x_2 > 10)) \wedge \neg g_4
\end{aligned}$$

which gives us $Sum_{main} = (g_0 \Rightarrow \neg(x_0 > 8)) \wedge \neg g_5$. The calling contexts $CallCtx_{bar} = \neg g_4$, $CallCtx_{foo_0} = g_2 \Rightarrow \neg(x_1 > 10)$, and $CallCtx_{foo_1} = g_3 \Rightarrow \neg(x_2 > 10)$ do not result in an update of the calling contexts for *foo* and *bar* (Line 8 in Alg. 3). $g_0 \wedge Sum_{main}$ is satisfiable, hence, $x \leq 8$ is a (maximal) refutation witness.

4.4.4.1 A note on (potentially) non-terminating programs

```
int main(int a){
  while(a){
    skip;
  }
  assert(0);
}
```

Figure 4.2

The example in Fig. 4.2 shows a potentially non-terminating program. Our technique is unaffected due to such non-termination as we restrict ourselves to recursion-free programs with a finite unwinding. Once unwound, this effectively reduces to the program shown in Fig. 4.3.

Note that there are no function calls in *main*, and $Props_{main} = (g_{in} \Rightarrow ((\neg a) \Rightarrow false))$. We use Eq. 4.5 to compute the summary of *main*.

$Sum_{main} \Rightarrow (\neg(Props_{main}))$, which reduces to

$Sum_{main} \Rightarrow (g_{in} \wedge \neg(a))$

Thus, the maximal summary of *main* w.r.t. the error being reachable is $g_{in} \wedge \neg(a)$. In other words, the assertion gets violated if *main* is reachable, and is executed with the input $a = 0$.

```
int main(int a){
  skip;
  assume (!a);
  assert(0);
}
```

Figure 4.3

4.5 Examples of Refutation Algorithms

Alg. 3' is not only applicable to loop-free programs with multiple procedure invocations, it can still be used for programs with loops by introducing invariants into the formula for the modular subproblem (4.5). However, in general it is hard to solve the problems without using approximations by bounding the number of unwindings and/or using abstractions for computing the predicates involved.

In Section 4.4, we have described the elements necessary for compositional, horizontal refutation proofs. In this section, we will give three examples of algorithms that instantiate this framework (Alg. 3'), which we have implemented to compare them experimentally in Section 4.6. We assume that loops have been unwound a finite number of times before application of these techniques. The difference in the following three techniques lies in the abstractions that are used to solve for Sum_f and $CallCtx_f$ in (4.5). We consider techniques that use constraint solving to find counterexamples.

4.5.1 Concrete Backward Interpretation

This technique is the one sketched in the example at the beginning of Section 4.3. Formally, we use the domain of predicates that track a single constant value for each variable, defined as follows: Let $P(\mathbf{x}) = \{false\} \cup \{\mathbf{x} = \mathbf{d} \mid d_i \in Dom(x_i)\}$ with the domain $Dom(x_i)$ of variable x_i, then we admit the following predicates for summaries and calling contexts: $Sum_f \in \{g_f^{in} \Rightarrow p \mid p \in P(\mathbf{x}^{in}{}_f)\}$ and $CallCtx_f \in \{g_f^{out} \Rightarrow p \mid p \in P(\mathbf{x}^{out}{}_f)\}$. We explain now in an example how Alg. 3' proceeds using this domain.

Example. Let us consider the example in Fig. 4.1 in Section 4.2. We start with $compose'(main, \neg g_5)$. We obtain the calling contexts $\neg g_2, \neg g_3, \neg g_4$ for foo_0, foo_1, bar, respectively. We recur into $compose'(bar, \neg g_9)$. We have to solve (4.5) where Sum_{bar}

is instantiated with the above domain:

$$\begin{array}{l} \exists d : \forall z, g_8, g_9 : \\ \quad (g_8 \Rightarrow z{=}d) \Longrightarrow \begin{array}{l} (\neg g_9 \vee \neg(g_8 \Rightarrow (z > 10)))\wedge \\ (g_9 = (g_8 \wedge z > 10)) \end{array} \end{array} \tag{4.6}$$

The partial order of our domain has only two levels *false* and the values for **d**. Hence, we can implement max by $\exists \mathbf{d}$; if there is no **d** then $p = \mathit{false}$. A constraint solver may return, for example, $d{=}{-}4$; $\mathit{Sum}_{\mathit{bar}}$ is hence $g_8 \Rightarrow (z{=}{-}4)$. This is an under-approximative summary of *bar* w.r.t. property violation.

In the next iteration of $\mathit{compose}'(\mathit{main}, \neg g_5)$ we solve:

$$\begin{array}{l} \exists d_0, \ldots, d_3 : \forall x_0, g_0, \ldots, g_5 : \\ \quad (g_0 \Rightarrow x_0{=}d_0)\wedge \\ \quad (g_2 \Rightarrow x_1{=}d_1)\wedge \\ \quad (g_3 \Rightarrow x_2{=}d_2)\wedge \\ \quad (g_4 \Rightarrow d_3) \Longrightarrow \begin{array}{l} (\neg g_5 \vee \neg \mathit{true})\wedge \\ g_1 = (g_0 \wedge (x_0 < 10))\wedge \\ g_5 = ((g_0 \wedge \neg(x_0 < 10)) \vee g_4)\wedge \\ \neg g_1 \wedge \neg g_2 \wedge \\ (g_3 \Rightarrow (x_2{=}{-}4)) \wedge \neg g_5 \end{array} \end{array} \tag{4.7}$$

and obtain $\mathit{CallCtx}_{\mathit{foo}_1} = (g_3 \Rightarrow (x_2{=}{-}4))$. $\mathit{compose}'(\mathit{foo}, g_7 \Rightarrow (r{=}{-}4))$ returns $g_6 \Rightarrow (y{=}{-}5)$ for $\mathit{Sum}_{\mathit{foo}_1}$. Note that the boolean variable d_3 stands for the reachability of the exit of *bar*. Since *bar* has no return value, this is how its exit is encoded. Proceeding similarly we get $\mathit{compose}'(\mathit{foo}, g_7 \Rightarrow (r{=}{-}5)) = (g_6 \Rightarrow (y{=}{-}6))$; and finally $\mathit{Sum}_{\mathit{main}} = (g_0 \Rightarrow x_0{=}{-}6)$. Hence, we have found a true global counterexample.

4.5.2 Abstract Backward Interpretation

Abstract backward interpretation computes sufficient preconditions to safety violations, i.e. negations of necessary preconditions to safety. The formula representing the summary may vary from being quite concise to as large as the procedure itself, depending on the abstraction.

There are a couple of techniques to implement such abstract interpretations that are distinguished by the way abstract preconditions are inferred, e.g. (classical) abstract domain transformers (e.g. [85]), template-based synthesis (e.g. [57]) or interpolation (e.g. [2]).

We are going to use the template-based synthesis technique used in [20] to solve (4.5). We know how to compute over-approximative abstractions with that technique. Hence, we use an over-approximation to compute an under-approximation (similar

to computing $\max f$ by $-\min(-f)$). This means we compute predicates $Sum_f^{\tilde{u}}$ and $CallCtx_j^{\tilde{u}}$ whose negations are Sum_f and $CallCtx_j$, respectively. This is done by solving the following formula in place of (4.5) in Alg. 3'.

$$\begin{aligned}
&\min Sum_f^{\tilde{u}}, \overbrace{CallCtx_j^{\tilde{u}}, \ldots}^{\text{for all } j \in CS_f} : \forall X_f : \\
&\quad \big(CallCtx_f^{\tilde{u}}(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge Sums_f^{\tilde{u}} \wedge \\
&\quad T_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge Props_f(\mathbf{x}_f) \implies \quad Sum_f^{\tilde{u}}(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge \\
&\qquad\qquad\qquad\qquad \textstyle\bigwedge_{j \in CS_f} CallCtx_j^{\tilde{u}}(\mathbf{x}^{p_in}{}_j, \mathbf{x}^{p_out}{}_j)\big)
\end{aligned} \tag{4.8}$$

This formula is derived from (4.5) by negating $(CallCtx_f \vee \neg Props)$ on the right-hand side of (4.5), which yields $(CallCtx_f^{\tilde{u}} \wedge Props)$, reversing the implication, and minimising to obtain an over-approximation for $Sum^{\tilde{u}}$ and $CallCtx^{\tilde{u}}$. Similar approaches are used in [28, 38]. Since convex domains are too imprecise for this purpose, we use a disjunctive domain [94]. For our experiments we used intervals as a base domain. Formally, let $P(\mathbf{x}) = \{\bigvee_k \mathbf{d}'_k \leq \mathbf{x} \leq \mathbf{d}_k \mid d_i, d'_i \in Dom(x_i), k \geq 0\}$, then $Sum_f \in \{g_f^{in} \Rightarrow p \mid p \in P(\mathbf{x}^{in}{}_f)\}$ and $CallCtx_f \in \{g_f^{out} \Rightarrow p \mid p \in P(\mathbf{x}^{out}{}_f)\}$. Our implementation also ensures that arithmetic overflows create new disjuncts in order to avoid precision loss. The second source of additional disjuncts that we take into account are Lines 7 and 12 in Alg. 3'.

Example. For the example in Fig. 4.1, we compute $compose'(main, \neg g_5)$. We solve (4.8) with $CallCtx_{main}^{\tilde{u}} = g_5$ and get $CallCtx_{bar}^{\tilde{u}} = g_4$, i.e. $CallCtx_{bar} = \neg g_4$.

We recur into $compose'(bar, \neg g_9)$, i.e. $CallCtx_{bar}^{\tilde{u}} = g_9$ We have to solve (4.8) instantiated with our domain.

$$\begin{aligned}
&\exists d, d' : \forall z, g_8, g_9 : \\
&\quad (g_9 \wedge true \wedge \\
&\quad (g_9 = (g_8 \wedge z > 10)) \wedge (g_8 \Rightarrow (z > 10)) \implies \quad (g_8 \Rightarrow (d \leq z \leq d')))
\end{aligned} \tag{4.9}$$

Note that $Sums_f^{\tilde{u}}$ is true because the initial under-approximations are false—the superscript $\tilde{u}$ flags predicates that carry negations of under-approximations. We get $Sum_{bar}^{\tilde{u}} = (g_8 \Rightarrow (11 \leq z \leq MAX))$, i.e. $Sum_{bar} = (g_8 \wedge (MIN \leq z \leq 10))$. MAX and MIN denote the maximum, resp. minimum, possible value for the type of z.

We proceed similarly. Finally, $compose'(main, \neg g_5)$ computes $Sum_{main}^{\tilde{u}} = (g_0 \Rightarrow (9 \leq x_0 \leq MAX))$, i.e. $Sum_{main} = (g_0 \wedge (MIN \leq x_0 \leq 8))$.

Note that (4.8) expresses an over-approximation of *good* states; the complement is therefore guaranteed not to contain any good states, but only *bad* and *unreachable* states, and hence no strict under-approximation of bad states. However, this does not

matter since we project $Sum_{f_{entry}}$ on the initial condition (see Thm. 4.4.2) to obtain a true under-approximation of inputs that violate a property.

Abstract backward interpretation is not limited to bounded unwindings of the transition relation, but can also be used for programs with loops (cf. [28,42]) by calling invariants into play in (4.8).

4.5.3 Symbolic Backward Interpretation

This technique computes the exact weakest precondition for the bounded problem. The technique is complete for loop-free programs. However, the size of the obtained summaries (i.e., size of the formula that represents the summary) may be of the order of the procedure size (i.e., size of the SSA of procedure's transition relation) in the worst case.

The domain used are sets of variables, so-called *dependency sets*. These sets of variables, X_f^{in}, X_f^{out}, $X_j^{p\_in}$, $X_j^{p\_out}$, describe which variables should be kept as relevant part of the summary. We then use them to compute an exact summary as the following predicate $Sum_f(\mathbf{x}^{in}, \mathbf{x}^{out})$:

$$\begin{aligned} &\exists X_f \setminus (X_f^{in} \cup X_f^{out} \cup \overbrace{X_j^{p\_in} \cup X_j^{p\_out} \cup \ldots}^{\text{for all } j \in CS_f}) : \\ &\quad (CallCtx_f(\mathbf{x}^{in}, \mathbf{x}^{out}) \vee \neg Props_f) \wedge T_f(\mathbf{x}^{in}, \mathbf{x}^{out}) \wedge Sums_f \end{aligned} \tag{4.10}$$

We implement the existential quantification in (4.10) by Gaussian elimination to eliminate as many of the intermediate or irrelevant variables as possible. After elimination the summary contains only variables that have a dependency on the property $Props_f$, on $\mathbf{x}^{out}$, or on the placeholder predicates, which are going to be replaced by summaries during the composition. The elimination can have positive and negative effects on the formula size depending on non-determinism and control flow paths in the procedure.

The composition operator is the same horizontal composition operator as in the two previous techniques. Context-sensitivity is exploited exactly in the same way as in the previous two techniques. The calling context at call site j is the set of output variables $X_j^{p\_out}$ that a procedure call backward-transitively depends on the given property. The resulting calling context dependency set X_f^{out} is then used for eliminating intermediate variables in (4.10) in addition to the dependency sets obtained from $Sums_f$, and $Props_f$. The set of input variables X_f^{in} that have not been eliminated is the dependency set $X_f^{p\_in}$ of the summary Sum_f.

Any satisfiable assignment to ${\mathbf{x}^{in}}_{f_{entry}}$ in the formula obtained by Gaussian elimination of the summary predicates in the entry function is a feasible global refutation.

Example. For example, in Fig. 4.1 the symbolic backward interpreter starts from the exit of *main* with $X^{out}_{main} = \emptyset$ to start with. As it arrives in *bar*, it retains the negation of the assertion $\neg(g_8 \Rightarrow (z > 10))$ and updates the dependency set to $X^{in}_{bar} = \{z, g_8\}$. On simplification, this gives the summary for *bar* as $g_8 \wedge \neg(z > 10)$.

Then the technique proceeds to the caller of *bar*, replacing the variables in the dependency set by the parameter passed, i.e. $X^{p\_out}_{foo_1} = \{x_2, g_3\}$. Then it recurs into the call to *foo*. The statement $r = y + 1$ gives the summary of *foo* as $r = y + 1$ and the dependency set $\{y, g_6\}$. The next call to *foo* has already been analysed with the same dependency set, hence there is no need to recur.

Proceeding in the main function, we finally get the summary for *main* as $(g_1 = g_0 \wedge (x_0 < 0)) \wedge foo_0((x_0, g_1), (x_1, g_2)) \wedge foo_1((x_1, g_2), (x_2, g_3)) \wedge bar((x_2, g_3), (g_4))$. Substituting the placeholder predicates by their respective summaries (variables are renamed) allows us to evaluate the summary for *main*. Since it is satisfiable, we have found a global refutation.

4.6 Experiments

We performed a number of experiments to evaluate compositional refutation techniques in comparison with monolithic approaches.

Implementation We have implemented these safety refutation techniques as an extension to 2LS [20, 96]. 2LS is a verification tool built on the CPROVER framework [36], using MiniSAT 2.2.1 as the backend solver (although other SAT and SMT solvers with incremental solving support can also be used). We limit resources to 900 seconds CPU time and 13 GB memory per benchmark. To aid reproducibility, we provide[5] the implementation sources along with the compilation instructions, the benchmarks, and scripts that can be used to run the tool on the benchmarks. As explained in Section 4.5, the three techniques are instances of a context-sensitive interprocedural analysis that traverses the callgraph backwards and propagates summaries and calling contexts. For the concrete interpretation, values for non-deterministic choices are picked by the SAT solver. For the abstract interpretation we use disjunctions of intervals. Note that termination is not an issue in this case because we are dealing with finite state systems. For infinite-state systems, one may use strategy iteration [20, 88]. We have implemented an algorithm based on strategy iteration, for finite-state systems.

[5] `https://github.com/kumarmadhukar/2ls/tree/atva17`

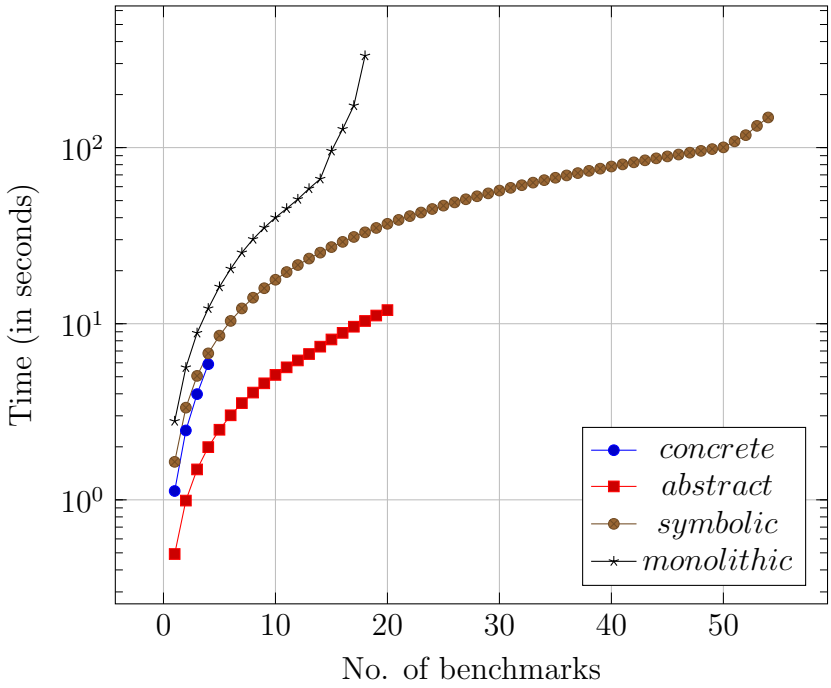


Figure 4.4: Comparison on Product Line benchmarks

Benchmarks We selected the unsafe examples (265 benchmarks) from the *product-lines* collection of the SV-COMP 2017 benchmarks set for our experiments. These benchmarks have a reasonably complex procedural structure (83 procedures per benchmark on average), which makes it suitable to test the effectiveness of our techniques. Our experiments were aimed at proving these benchmarks unsafe, i.e. for every benchmark program, we stopped at the first assertion violation that was reached. We set an unwinding depth of 5 for all the benchmarks, across all the techniques. The chosen depth might have been, in some cases, higher than what would be necessary to find a refutation. However, the aim of our experiments was to compare the scalability of the techniques in general, and not to find out the least amount of time needed to decide if a given benchmark is unsafe.

Results Fig. 4.4 shows the results plotting for each technique the cumulative time (y-axis) it takes to solve (i.e., to decide that it is unsafe) the given number of benchmarks (x-axis). The longer the line for a technique extends to the right the more benchmarks were solved within the resource limits. These results show some interesting tendencies. We observe that the symbolic backward interpretation performs best. It is complete, but could potentially degrade into a monolithic analysis if summaries cannot be sufficiently simplified and reused. But on this benchmark set it works quite well on

a certain number of benchmarks. The abstract backward interpretation is very fast on a couple of benchmarks, but then remains inconclusive. This is supposedly due to the imprecision introduced by the weak abstract domain that we use. Yet, this is encouraging that by a clever choice of abstractions one could outperform the symbolic backward interpretation. The concrete backward interpretation succeeds only on very few benchmarks and is surprisingly slow. An explanation for this is that it is required to make non-deterministic choices that may turn out to be bad choices and make a counterexample infeasible. Moreover, the summaries that it computes usually do not generalise beyond the procedure invocation they were generated for. Hence, this technique is likely to degrade into following the entire execution path, spoiling the benefits of modularity while exhibiting the drawbacks of abstraction. The monolithic analysis (bounded model checking), which is based on full inlining is slowest but solves almost as many benchmarks as the abstract one.

4.7 Extension to Loops

This section is an extension of the refutation problem to non-recursive programs with loops. The formalisation given here introduces inductive invariant predicates (see Section 4.2), besides summaries in calling contexts, to abstract effect of loops in the program.

Formalizing the Input/Output Transition System To tackle programs with loops, we first formalize the input/output transition system a little differently from what we have seen earlier. Let the input/output transition system of a procedure f be a triple $(Init_f, T_f, Out_f)$, where $T_f(\mathbf{x}, \mathbf{x}')$ is the transition relation, which can encode both, loop-free procedures as well as procedures with loops; the input relation $Init_f(\mathbf{x}^{in}, \mathbf{x})$ defines the initial states of the transition system and relates it to the inputs $\mathbf{x}^{in}$; the output relation $Out_f(\mathbf{x}, \mathbf{x}^{out})$ connects the transition system to the outputs $\mathbf{x}^{out}$ of the procedure. Inputs, as earlier, are procedure parameters, global variables, and memory objects that are read by f. Similarly, outputs are return values, and potential side effects such as global variables and memory objects written by f. Internal states $\mathbf{x}$ are commonly the values of variables at the loop heads (if any) in f. Modeling T for procedures with loops is done by simply modeling each loop body as a piece-wise transition relation. The transition relation for the entire procedure is obtained by stitching the piece-wise transition relations using boolean guard variables (g) to model control flow and continuation.

Definition 4.7.1 (Invariants)**.** *For a procedure given by* $(Init_f, T_f, Out_f)$ *we define:*

- *An* invariant *of* f *is a predicate* Inv_f *such that:*

$$\begin{aligned} \forall \mathbf{x}^{in}, \mathbf{x}, \mathbf{x}' : \quad & \big(Init_f(\mathbf{x}^{in}, \mathbf{x}) \implies Inv_f(\mathbf{x})\big) \\ \wedge \quad & \big(Inv_f(\mathbf{x}) \wedge T_f(\mathbf{x}, \mathbf{x}') \implies Inv_f(\mathbf{x}')\big) \end{aligned}$$

Monolithic safety refutation The following formula extends the monolithic safety verification problem from (4.3) to include invariants.

$$\begin{aligned} &\exists \overbrace{Sum_f, Inv_f, CallCtx_f, \ldots :}^{\text{for all } f \in F} \textstyle\bigwedge_{f \in F} \forall X_f : \\ &\quad CallCtx_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \implies \\ &\quad\quad \big(Init_f(\mathbf{x}^{in}{}_f, \mathbf{x}_f) \implies Inv_f(\mathbf{x}_f)\big) \\ &\quad \wedge \big(Inv_f(\mathbf{x}_f) \wedge T_f(\mathbf{x}_f, \mathbf{x}'_f) \wedge \\ &\quad\quad Sums_f \qquad\qquad\qquad \implies Props_f(\mathbf{x}_f) \wedge Inv_f(\mathbf{x}'_f) \wedge \\ &\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textstyle\bigwedge_{j \in CS_f} CallCtx_{func(j)}(\mathbf{x}^{p\_in}{}_j, \mathbf{x}^{p\_out}{}_j)\big) \\ &\quad \wedge \big(Init_f(\mathbf{x}^{in}{}_f, \mathbf{x}_f) \wedge Inv_f(\mathbf{x}_f) \wedge \\ &\quad\quad Inv_f(\mathbf{x}'_f) \wedge Out_f(\mathbf{x}'_f, \mathbf{x}^{out}{}_f) \quad \implies Sum_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f)\big) \end{aligned} \tag{4.11}$$

Here, X_f refers to the variables in T_f, including intermediate and primed variables.

The inputs and outputs are constrained by the calling context $CallCtx_f$; the first two conjuncts are the base and step case to define inductiveness of the invariant, and the last conjunct defines the summary.

Modular safety refutation We use the same approach as in (4.4) to derive a modular formulation. I.e. for each $f \in F$ we have:

$$\begin{aligned} &\min Sum_f, Inv_f, \overbrace{CallCtx_j, \ldots :}^{\text{for all } j \in CS_f} \forall X_f : \\ &\quad CallCtx_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \implies \\ &\quad\quad \big(Init_f(\mathbf{x}^{in}{}_f, \mathbf{x}_f) \implies Inv_f(\mathbf{x}_f)\big) \\ &\quad \wedge \big(Inv_f(\mathbf{x}_f) \wedge T_f(\mathbf{x}_f, \mathbf{x}'_f) \wedge \\ &\quad\quad Sums_f \qquad\qquad\qquad \implies Props_f(\mathbf{x}_f) \wedge Inv_f(\mathbf{x}'_f) \wedge \\ &\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textstyle\bigwedge_{j \in CS_f} CallCtx_{func(j)}(\mathbf{x}^{p\_in}{}_j, \mathbf{x}^{p\_out}{}_j)\big) \\ &\quad \wedge \big(Init_f(\mathbf{x}^{in}{}_f, \mathbf{x}_f) \wedge Inv_f(\mathbf{x}_f) \wedge \\ &\quad\quad Inv_f(\mathbf{x}'_f) \wedge Out_f(\mathbf{x}'_f, \mathbf{x}^{out}{}_f) \quad \implies Sum_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f)\big) \end{aligned} \tag{4.12}$$

The composition operator is Alg. 3.

To obtain witnesses, we extend (4.5) to include invariants.

$$\begin{aligned}
&\max Sum_f, Inv_f, \overbrace{CallCtx_j, \ldots}^{\text{for all } j \in CS_f} : \forall X_f : \\
&\quad \big(Inv_f(\mathbf{x}_f) && \implies (CallCtx_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge Out(\mathbf{x}^{out}{}_f, \mathbf{x}_f) \vee \\
&\qquad && \quad \neg Props_f)\big) \\
&\quad \wedge \big(Inv_f(\mathbf{x}_f) \wedge \\
&\qquad \textstyle\bigwedge_{j \in CS_f} CallCtx_j(\mathbf{x}^{p_in}{}_j, \mathbf{x}^{p_out}{}_j) && \implies Inv_f(\mathbf{x}'_f) \wedge T_f(\mathbf{x}_f, \mathbf{x}'_f) \wedge Sums_f\big) \\
&\quad \wedge \big(Sum_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) && \implies CallCtx_f(\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f) \wedge Init_f(\mathbf{x}^{in}{}_f, \mathbf{x}_f) \wedge \\
&\qquad && \quad Inv_f(\mathbf{x}_f) \wedge Inv_f(\mathbf{x}'_f) \wedge Out_f(\mathbf{x}'_f, \mathbf{x}^{out}{}_f)\big)
\end{aligned} \tag{4.13}$$

As above, the first and second conjuncts are the inductive definition of the invariant, and the last conjunct is the definition of summary. Note that the base case is slightly different because we start from property violations that might either come from the calling context or the properties in f itself. Property violations in the callees are considered by the step case. The composition operator is Alg. 3'. Note that the algorithm does not synthesize invariants itself; it assumes that the invariants are supplied from an external engine. The refutation algorithms with varying degrees of completeness (discussed in Section 4.5) would continue to work in this case as well, if we assume that the loops have been replaced with the given loop invariants.

```
void main()
{
  int x;
  while(x < 2){
    x++;
    foo(x);
  }
}

void foo(int y)
{
  assert(y < 1);
}
```

$$\begin{aligned}
T_{main}((x_0, g_0), (g_3)) &\equiv g_1{=}(g_0 \wedge (x_0{<}2)) \wedge \\
&\quad\ (x_1{=}x_0{+}1) \wedge \\
&\quad\ foo_0((x_1, g_1), (x_2, g_2)) \wedge \\
&\quad\ g_3{=}((g_0 \wedge \neg(x_0{<}2)) \vee \\
&\qquad (g_2 \wedge \neg(x_1{<}2))) \\
Props_{main} &\equiv true \\
\\
T_{foo}((y, g_4), (g_5)) &\equiv g_5{=}(g_4 \wedge (y{<}1)) \\
Props_{foo} &\equiv g_4 \Rightarrow (y{<}1)
\end{aligned}$$

Figure 4.5: Example program with loop and its encoding

For example, let us apply the *Abstract Backward Interpretation* technique 4.5.2 to the program shown in Fig. 4.5. Proceeding similarly as in the loop-free case, we get $Sum^{\tilde{u}}_{foo} = (g_4 \Rightarrow (MIN \leq y \leq 0))$, i.e. $Sum_{foo} = (g_4 \wedge (1 \leq y \leq MAX))$. Moreover, at the beginning of the while loop in *main*, $(MIN \leq x \leq 1)$ is an invariant, assuming that the input x satisfies $x \leq 1$ when *main* is called. Note that x may get the value 2 inside the loop, but the subsequent call to *foo* would ensure that an assertion violation is reached and the value 2 never gets propagated at the loop-head. Therefore, using

this invariant and Sum_{foo}, we deduce that an assertion violation is reachable if x may lie in the range $[0, 1]$ at the loop-head. Clearly, this is possible for all inputs x that are in the range $[MIN,1]$. Thus, we get the maximal refutation summary for *main* as $Sum_{main} = (g_0 \wedge (MIN \leq x_0 \leq 1))$. In other words, the property is refutable if the initial value of x is in the range $[MIN, 1]$.

Note that we have used loop invariants to replace loops in the program. In general, this may be done only if loop invariants are precise (and not over-approximate as they could typically be) to guarantee soundness.

4.8 Discussion on Related Work

Compositional automated verification approaches have been considered in the tools Whale [2] and FunFrog [99], for example. Horn clause encodings were used in [71]. These tools eventually use interpolation to compute abstractions. Under-approximating precondition inference techniques have been proposed for polyhedra [85] and with the help of bit blasting and loop iteration estimation [23]. All these techniques can be used in our setting, however, their completeness properties remain to be evaluated. Completeness considerations [89] have been conducted for compositional LTL model checking [33, 35] of (parallel) compositions of (infinite-) state transition systems. Since the decomposition of sequential programs can be encoded into a composition of parallel programs (with appropriate synchronisation), their completeness results are expected to hold in our setting. Compositionality has also been explored in the context of dynamic test generation to achieve scalability by memoizing symbolic execution sub-paths as test summaries [53]. This has given rise to an incremental approach for statically validating symbolic test summaries against code changes [54]. In our framework memoization is naturally handled by the composition operator.

4.9 Concluding Remarks

We investigated compositional refutation techniques in horizontal, e.g. procedure-modular, decompositions of sequential programs. We showed how to derive a compositional refutation framework step by step from the monolithic problem. We also compared the completeness properties of concrete, abstract and symbolic modular refutation approaches. Our experiments show that compositional refutation techniques have an advantage over monolithic approaches, however, not all tested approaches perform equally well because of their varying completeness.

```
int main(){
  int x = *;
  foo(x);
  return 0;
}

int foo(int a){
  if(a < 10){
    a = bar(a);
    a = bar(a);
    baz(a);
  }
  return 0;
}

int bar(int b){
  return b+1;
}

int baz(int c){
  assert(c > 16);
  return 0;
}
```

Figure 4.6: An example program to demonstrate benefits of the compositional approach

There are two reasons why the compositional approach is advantageous. First, while traversing backward, from an assertion, the spuriousness may often be discovered without having to go all the way back to the entry point. In fact, reaching the entry point (from assertion, backward) would mean an actual safety refutation. In the example shown in Fig. 4.6, $c = 14$ is a counterexample obtained locally for the function *baz* but it is spurious for the entire program. This spuriousness can be detected at *foo* itself when the counterexample is propagated backward. Secondly, the summaries computed once may be cached and reused, under suitable calling contexts. For example, in the program shown above, the summary of *bar* may be reused when it is called for the second time.

Using a portfolio of fast incomplete techniques and slower complete ones may ensure that modular techniques are always at least as fast as monolithic ones in practice.

Open questions Modular analyses should be independent of a program's syntactic structure because real-world programs are not written in a nice and balanced way that would enable efficient modular analysis. It would be worthwhile to explore semantic decompositions into modules in order to make these techniques scale on real-world programs. W.r.t. the inter-procedural backward analysis, it remains to be investigated how to handle recursion.

Moreover, it would be interesting to look into compositional refutation in termination analysis. Also there, spuriousness of local refutations can occur due to lack of context information: To find a counterexample to termination one needs to find a stem from the entry point. Compositionality in this context has been explored in the Ultimate tool [64]. We would also consider performance comparisons with testing, i.e. dynamic refutation techniques (random, directed, concolic, etc.) to be beneficial to advance research in static refutation techniques.

4.9.1 Notes

The techniques that we have developed for safety refutation can be useful in developing a framework that handles both safe and unsafe programs. In particular, the refutation techniques may be thought of as one part of a bigger cycle, of abstraction-refinement. This would allow one to work with both under-approximations and over-approximations, and refine both of them as required. The under-approximations can be refined by moving to a more complete spuriousness check algorithm, as discussed in this chapter. In the next one, we sketch the larger framework, by combining compositional verification, inter-procedural analysis, and k-induction.

Chapter 5

Exploiting Modularity of Implementation: Proofs

Refutation techniques discussed in the previous chapter allow us to find safety violations compositionally. But what if there are no violations to the safety property, i.e. the property cannot be refuted? In other words, what if the program is *safe*? This chapter discusses a refinement based algorithm, using k-induction, to obtain a *proof* of safety for such programs.

Fig. 5.1 shows the generic structure of a refinement algorithm framework for verification. As per this framework, one may begin the search for a proof (or a counterexample) by starting with an over-approximation, even a coarse one e.g. *true*. An over-approximation may be obtained in various ways, e.g. by replacing a called procedure with an over-approximating summary, by over-approximating the context in which a procedure is invoked, etc. Since we assume that loops have been unwound to a fixed depth, this over-approximation is actually that of an under-approximation of the program, and not of the original program itself. If, at any point during the verification, it gets established that an over-approximation is safe (or unsafe, with a non-spurious counterexample) then the original program is also safe (unsafe), and the search aborts. Otherwise, one may switch to an under-approximation of the program, and look for counterexamples. If a counterexample is found, the original program is declared unsafe. Otherwise, a refinement is needed. In this framework, the refinement for over-approximations is unwinding the program and inlining the procedure calls. The under-approximation, on the other hand, is refined by using a more complete algorithm for checking the spuriousness of the abstract counterexample. This happens in the cycle starting from *under-approximation*, to *check properties*, *cannot decide*, *refine* and back to a refined *under-approximation* (shown on the right in Fig. 5.1).

Practical implementations of the refinement procedures are usually incomplete, which leads to the extension of the algorithm by the dashed elements in Fig. 5.1. On failure to refine the under-approximation, one can still try to refine the over-approximation in the hope that this refinement helps avoiding the "cannot decide" situation in the next iteration. On a subsequent failure to refine the over-approximation one is forced to give up.

The refinement box on the right could be a more *complete* algorithm for spuriousness check (as described in Section 4.5), while the one on the left could be inlining or strengthening of summaries or even unwinding. Though, traditionally, unwinding is not viewed as a refinement strategy when doing bounded model checking, in Fig. 5.1 the refinement of over-approximation on the left may be done by using a larger unwinding.

One could use an entirely different approach for proving correctness, by explicitly finding an inductive assertion that strengthens the property under consideration. This may be done through various ways such as abstract interpretation, recurrence analysis, interpolation, dynamic analysis, learning, etc. Each technique has its own challenges and limitations. In contrast to these, Bounded Model Checkers exploit the finiteness of the state graph to enable a complete approach for proving safety, based on unrolling the transition relation. We explore a similar approach in this chapter as it naturally fits into our framework, however we do not wish to rely on a property of the state graph. Therefore, we use k-induction. It is a technique that allows verification to succeed using weaker loop invariants, than are required, by strengthening the premise sufficiently so that an inductive argument may successfully be made. This strengthening is done by considering multiple steps of the transition relation at once (for instance, by unrolling loops partially and adding the unrolled body to the already existing premise). Informally, in the framework described in Fig. 5.1 above, the base case check happens on the right hand side (using under-approximations), while the inductiveness check happens on the left. Although k-induction is one of the most popular techniques for proving safety, for large programs, the bounded model checking instances generated to check k-inductive proofs often exceed the limits of resources available. Performing k-induction in a modular way could speed up verification. We propose an interprocedural approach to modular k-induction, as an instance of a more general refinement approach to program verification.

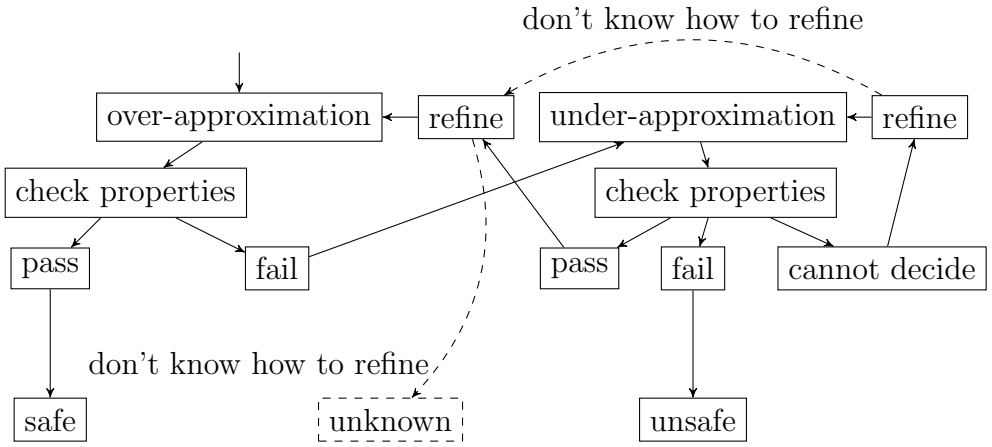


Figure 5.1: Generic Structure of a Refinement Algorithm Framework

5.1 Motivating a Compositional Approach

Despite recent advent of several promising model-checking algorithms [3, 13, 82] for safety verification of software, scaling it to real world programs beyond a few thousand lines of code remains a serious challenge. Bounded model checking (BMC) [36] scales better, but is usually incomplete in practice. k-induction extends BMC-based approaches from falsification to verification. Nevertheless, its scalability remains an issue as the value of k may become quite large (it increases in every iteration until a safety proof is obtained for safe programs), resulting in the BMC instances becoming unmanageable.

One of the main restrictions that limits the scalability of these techniques is that they are not *compositional*. That is, they analyze code as a monolithic usually flattened entity instead of using a divide-and-conquer approach that exploits the syntactic or semantic structure, e.g. procedure hierarchy, in the program. For example, BMC inlines all procedures in the program before unwinding the loops. Most leading complete model-checking methods for safety verification [3, 13, 20, 82], including ones that use over-approximations, are not compositional. In this work, we develop a compositional k-induction technique for safety verification of software. A central problem that a compositional approach for k-induction brings in is that of selective refinement (be it unwinding or inlining), which has been addressed in this chapter.

A decomposition of a verification problem intuitively splits the original problem into a set of subproblems that cover the original problem. The decomposition operator for the problem has a corresponding composition operator for composing the results obtained from the subproblems in order to obtain the solution of the original problem.

Compositionality has been naturally studied in the context of the parallel composition of processes (e.g. [33, 35, 89]) where the decomposition is performed according the the process structure and the composition operator is a rely-guarantee proof rule, for example.

For sequential programs, decompositions can be *horizontal*, e.g. procedure-modular decompositions. In terms of program executions, a horizontal decomposition cuts execution traces into pieces, i.e. each element of the decomposed program captures a set of subtraces. *Vertical* decomposition focuses on whole execution traces. E.g. program slices (e.g. [62]) are an example of vertical decompositions.

5.1.1 Contributions

We summarise the contributions of this chapter as follows.

- We begin with an informal overview of our technique (Section 5.4), and then formulate interprocedural verification by k-induction (Section 5.5).
- We propose a horizontal, i.e. interprocedural k-induction approach based on an interprocedural counterexample spuriousness check (Section 5.5.1) and a selective refinement of loop unwindings and procedure inlining (Section 5.5.2).

5.2 Discussion on Related Work

The novelty of our work lies in connecting three well-studied techniques - k-induction, compositional verification, and inter-procedural analysis. Hence we can only give a brief overview of the vast amount of relevant literature.

Since it was observed [100] that *k-induction* for finite state systems (e.g. hardware circuits) can be done by using an (incremental) SAT solver [45], it has become more and more popular also in the software community as a tool for safety proofs. Using SMT solvers, it has been applied to Lustre models [59] (monolithic transition relations) and C programs [44] (multiple and nested loops).

k-induction often requires additional invariants to succeed, which can be obtained by abstract interpretation. For example, Garoche et al. [50] use SMT solving to infer intermediate invariants over templates for the use in k-induction of Lustre models. As most of these approaches (except [22]), they consider (linear) arithmetic over rational numbers only, whereas our target are C programs with bit-vectors (representing machine integers, floating-point numbers, etc). In [20], k-induction framework was integrated with a template-driven SMT-based loop invariants generation method.

This integrated framework exploited k-induction as a refinement strategy for deriving stronger invariants and using the derived invariants in turn to strengthen induction hypothesis for improve success of an induction proof. The work presented in this chapter can be considered as a compositional version of that work by using procedure summaries derived via integration of an inter-procedural analysis technique.

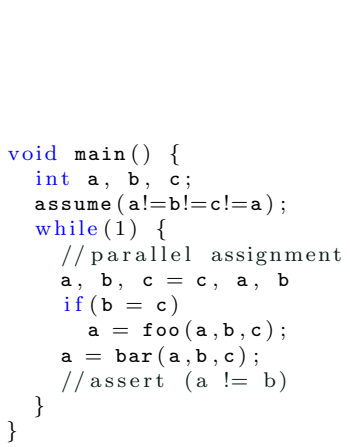
The idea of synthesizing abstractions with the help of solvers can be traced back to predicate abstraction [55]; Reps et al. [91] proposed a method for symbolically computing best abstract transformers; these techniques were later refined [22, 70, 102] for application to various template domains. Using binary search for optimization in this context was proposed by Gulwani et al. [57]. Similar techniques using LP solving for optimization originate from strategy iteration [52]. Recently, SMT modulo optimization [78, 98] techniques were proposed that foster application to invariant generation by optimization. While all of these works show how to perform compositional proofs using abstractions and invariants, the missing pieces are how to check for a spurious counterexample and how to refine the abstractions once a spurious counter example is discovered. We take a small step towards plugging this gap. Our refinement step, through selective unwinding and inlining of procedure calls, delays the construction of an exponentially-sized formula as much as possible, similar to the stratified inlining proposed by Lal et al. [75].

An alternative to alleviating the state-space explosion problem compositionally is through assume-guarantee reasoning [69,87,90]. Under this framework, each component of a system makes an assumption about the behaviour of other components, and, in return, guarantees something about its own behaviour. However, generating proper assumptions remains to be a major issue in the practicability of this.

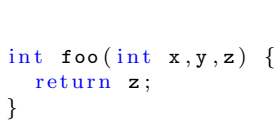
Interpolation-based algorithms have also been used, in an inter-procedural setting, to provide modular safety proofs of sequential programs (e.g. Whale [2, 71]). Whale uses Craig interpolants to compute function summaries by generalizing from under-approximations of functions. In contrast, our work uses k-induction and involves a refinement loop instead of generalization.

5.3 Preliminaries

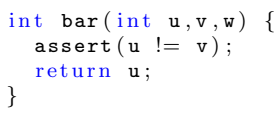
Program model and notation. The program model and the notation that we use here is the same as in the last chapter (see Sections 4.7 and 4.2). We recall that we view programs are symbolic transition systems. Its states are described by an interpretation to the program variables, and formulas may be used to describe a set of



$$
\begin{aligned}
Init_{main}((), (a_0, b_0, c_0, g_0)) &\equiv (a_0 \neq b_0 \neq c_0 \neq a_0) \wedge g_0 \\
T_{main}((a_0, b_0, c_0, g_0), & \\
(a_3, b_1, c_1, g_6)) &\equiv (g_1 = g_0 \wedge true) \wedge \\
& \quad a_1 = c_0 \wedge b_1 = a_0 \wedge c_1 = b_0 \wedge \\
& \quad g_2 = (g_1 \wedge (b_1 = c_1)) \wedge \\
& \quad g_2 \Rightarrow \\
& \quad\quad foo_0(((a_1, b_1, c_1), g_2), (a_2, g_3))) \wedge \\
& \quad g_4 = (g_3 \vee \neg(g_2)) \wedge \\
& \quad bar_0(((a_2, b_1, c_1), g_4), (a_3, g_5)) \wedge \\
& \quad g_6 = (g_5 \vee \neg(g_1')) \\
Out_{main}() &\equiv true \\
Props_{main} &\equiv true
\end{aligned}
$$



$$
\begin{aligned}
Init_{foo}((x, y, z), & \\
(x_0, y_0, z_0, g_7)) &\equiv x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge g_7 \\
T_{foo}(((x_0, y_0, z_0), g_7), & \\
(z_1, g_8)) &\equiv g_8 = g_7 \\
Out_{foo}(((x_0, y_0, z_0), g_7), & \\
(z_1, g_8)) &\equiv g_8 \Rightarrow (z_1 = z_0) \\
Props_{foo} &\equiv true
\end{aligned}
$$



$$
\begin{aligned}
Init_{bar}((u, v, w), & \\
(u_0, v_0, w_0, g_9)) &\equiv u = u_0 \wedge v = v_0 \wedge w = w_0 \wedge g_9 \\
T_{bar}(((u_0, v_0, w_0), g_9), & \\
(u_1, g_{10})) &\equiv g_{10} = (g_9 \wedge \neg(u_0 = v_0)) \\
Out_{bar}(((u_0, v_0, w_0), g_9), & \\
(u_1, g_{10})) &\equiv g_{10} \Rightarrow (u_1 = u_0) \\
Props_{bar} &\equiv (u_0 \neq v_0)
\end{aligned}
$$

Figure 5.2: Example program and its encoding

states. T is a transition relation between pairs of such interpretations which describes how the states are transformed by the program. Fig. 5.2 gives an example of the encoding of a program into such formulae. Before we proceed, note that one may attempt to find an inductive and adequate invariant for such programs, and thereby prove the program safe. An inductive invariant is one that holds in the beginning (the *initial* state), and assuming that it holds in an arbitrary state, may be proved to hold in the subsequent state as per the transition relation. The notion of adequacy is only with respect to the property, i.e. it is adequate if it can discharge the property.

But discovering an adequate inductive invariant is an orthogonal task to what we are trying here. One may think of these as two ends of a spectrum. On the one end,

we speak of fixing k to 1, and search for an inductive strengthening of the property. While on the other end, k-induction fixes the property to be what it is, and looks for a k such that it is k-inductive. Further, generating an adequate invariant is a difficult task. Hence, it certainly makes sense to look at techniques at the opposite end of the spectrum, or even a combination of the two extremes. The inputs $\mathbf{x}^{in}$ of *foo* are (x, y, z), and the outputs $\mathbf{x}^{out}$ consist of the return value of *foo*, denoted by z_1 in the program. The transition relation, T_{foo}, encodes the procedure body over the internal state variables (x_0, y_0, z_0). If a procedure modifies the state variables multiple times, such as in *main*, then we may need more than the initial variables to model the T formula using SSA formula to denote each update to a variable. For the procedure *main* shown in Fig. 5.2, T_{main} essentially contains the transition relation for the body of the while loop. The loop control is modeled by additional loop control guard variables, e.g. g_1 in *main*. The guard g_1 tell us if the body of the loop in *main* is reachable or not. The loop body is reachable if and only if *a)* the procedure *main* is reachable (modeled by the guard g_0), and *b)* the loop condition (which is simply *true* in this case). This is why we see the expression $(g_1 = g_0 \wedge true)$ in T_{main}. The post loop control variable (denoted by a primed version of the corresponding variable) is set to *false* to denote exiting from the loop. In procedure *main* in Fig. 5.2, the placeholder for the first procedure call to *foo* is $foo_0(((a_1, b_1, c_1), g^{in}_{foo}), (a_2, g^{out}_{foo}))$ with the actual inputs and output parameters and the entry/exit guards.

5.3.1 Monolithic k-Induction

As we know, bounded model checking (BMC) focuses on refutation by picking an unwinding limit k and solving:

$$\exists \mathbf{x}_0, \ldots, \mathbf{x}_k : \quad Init(\mathbf{x}^{in}, \mathbf{x}_0) \wedge \mathcal{T}[k] \wedge \neg \mathcal{P}[k+1] \tag{5.1}$$

where

$$\mathcal{T}[k] = \bigwedge_{i \in [0,k-1]} T(\mathbf{x}_i, \mathbf{x}_{i+1}) \qquad \mathcal{P}[k] = \bigwedge_{i \in [0,k-1]} \neg Err(\mathbf{x}_i)$$

and the predicate *Err* denotes an error state, i.e. a state violating *Props*.

Incremental BMC avoids the need for a fixed bound by repeatedly using BMC with increasing bounds, often optimized by using the solver incrementally. If the bound starts linearly from zero, it may be assumed that there are no errors in the previous states giving a simpler test:

$$\exists \mathbf{x}_0, \ldots, \mathbf{x}_k : \quad Init(\mathbf{x}^{in}, \mathbf{x}_0) \wedge \mathcal{T}[k] \wedge \mathcal{P}[k+1] \wedge Err(\mathbf{x}_k) \tag{5.2}$$

k-induction [100] can be thought of as an extension of incremental BMC that can show system safety as well as produce counterexamples. It makes use of k-inductive invariants, which are predicates, $KInv$, for which the following holds:

$$\forall \mathbf{x}_0 \ldots \mathbf{x}_k : \mathcal{I}[k] \wedge \mathcal{T}[k] \Rightarrow KInv(\mathbf{x}_k) \tag{5.3}$$

where

$$\mathcal{I}[k] = \bigwedge_{i \in [0,k-1]} KInv(\mathbf{x}_i) \tag{5.4}$$

k-inductive invariants have the following useful properties:

- Any inductive invariant is a 1-inductive invariant and vice versa.
- Any k-inductive invariant is a $(k+1)$-inductive invariant.
- A system is safe if and only if there is a predicate, $KInv$, which satisfies:

$$\begin{aligned} \forall \mathbf{x}_0 \ldots \mathbf{x}_k : \quad & (Init(\mathbf{x}^{in}, \mathbf{x}_0) \wedge \mathcal{T}[k] \Rightarrow \mathcal{I}[k]) \wedge \\ & (\mathcal{I}[k] \wedge \mathcal{T}[k] \Rightarrow KInv(\mathbf{x}_k)) \wedge \\ & (KInv(\mathbf{x}_k) \Rightarrow \neg Err(\mathbf{x}_k)) \end{aligned} \tag{5.5}$$

In this formula, the first two conjuncts ensure that $KInv$ is a k-inductive invariant. The first conjunct confirms that $KInv$ holds in the first k-steps of the program (i.e., the *base-case* necessary for the inductive argument to work). The second one guarantees that if $KInv$ holds for any sequence of k-steps, then it holds in the $(k+1)^{th}$-step. The last conjunct says that $KInv$ is sufficient to establish that the program is safe.
According to Brain et al. [20,21], showing that a k-inductive invariant exists is sufficient to show that an inductive invariant exists but it does not imply that the k-inductive invariant is an inductive invariant. Often the corresponding inductive invariant is significantly more complex. Thus, k-induction can be seen as a trade-off between generating invariants, and checking them, as it is a means to benefit as much as possible from simpler invariants by using a more complex property check.

Finding a candidate k-inductive invariant is hard so implementations often use $\neg Err(\mathbf{x})$. Linearly increasing k can be used to simplify the expression by assuming there are no errors at previous states:

$$\begin{aligned} \exists \mathbf{x}_0, \ldots, \mathbf{x}_k : \quad & (Init(\mathbf{x}^{in}, \mathbf{x}_0) \wedge \mathcal{T}[k] \wedge \mathcal{P}[k] \wedge Err(\mathbf{x}_k)) \vee \\ & (\mathcal{T}[k] \wedge \mathcal{P}[k] \wedge \neg Err(\mathbf{x}_k)) \end{aligned} \tag{5.6}$$

A model of the first part of the disjunct is a concrete counterexample (k-induction subsumes bounded model checking) and if the whole formula has no models, then $\neg Err(\mathbf{x})$ is a k-inductive invariant and the system is safe.

Applying monolithic k-induction to a program with multiple procedures effectively involves constructing T corresponding to the flattened program obtained by inlining all the procedure calls by their body instances. For example, the monolithic version of T for *main* in Fig. 5.2 is obtained by inlining the call instances to foo and *bar* by their instantiated procedure bodies, and then constructing a T and $Init$ for the inlined *main*. The goal of this work is to devise an algorithm to perform k-induction in a compositional way using summaries for the procedure calls using inlining only as required.

Theorem 5.3.1 (Monolithic k-induction)**.**
i) If the property is k-inductive for some k then the monolithic k-induction algorithm will prove it.
ii) If there is a counterexample reachable after k iterations, then the monolithic k-induction algorithm will find it.

Proof (sketch): Since we are talking about finite state systems, if the program is safe, $\neg Err(\mathbf{x})$ is itself k-inductive for some k ($\leq n$). This holds true because if the program is safe, then $Err(\mathbf{x})$ cannot intersect with the set of reachable states, and the latter can be precisely computed when the program is fully unrolled (i.e., when $k = n$). Thus, since k is incremented in each step, the proof is bound to succeed sometime unless a counterexample is obtained first.

In case of an unsafe program, where a counterexample is reachable in k-iterations, the monolithic k-induction will eventually reach that k and obtain a counterexample (because equation 5.5 would get violated).

5.3.2 Interprocedural Analysis

Moving on to interprocedural analysis, we introduce formal notation for the basic concepts below:

Definition 5.3.2. *For a procedure given by ($Init$, T, Out) we define:*

- *An* invariant *is a predicate Inv such that:*

$$\begin{aligned} \forall \mathbf{x}^{in}, \mathbf{x}, \mathbf{x}' : \quad & Init(\mathbf{x}^{in}, \mathbf{x}) \implies Inv(\mathbf{x}) \\ \wedge \quad & Inv(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \implies Inv(\mathbf{x}') \end{aligned}$$

- *Given an invariant Inv, a* summary *is a predicate Sum such that:*

$$\forall \mathbf{x}^{in}, \mathbf{x}, \mathbf{x}', \mathbf{x}^{out} : \quad Init(\mathbf{x}^{in}, \mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \wedge Inv(\mathbf{x}') \wedge Out(\mathbf{x}', \mathbf{x}^{out}) \implies Sum(\mathbf{x}^{in}, \mathbf{x}^{out})$$

- *Given an invariant Inv, the* calling context *for a procedure call h at call site i in the given procedure is a predicate* $CallCtx_{h_i}$ *such that*

$$\forall \mathbf{x}, \mathbf{x}', \mathbf{x^{p\_in}}_i, \mathbf{x^{p\_out}}_i : \\ Inv(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}') \implies CallCtx_{h_i}(\mathbf{x^{p\_in}}_i, \mathbf{x^{p\_out}}_i)$$

These concepts have the following roles. Invariants abstract the behaviour of loops. Summaries abstract the behaviour of called procedures; they are used to strengthen the placeholder predicates. Calling contexts abstract the caller's behaviour w.r.t. the procedure being called. When analyzing the callee, the calling contexts are used to constrain its inputs and outputs.

In Fig. 5.2, a candidate $Inv(a, b, c)$ for the loop in *main* is $(a \neq b \neq c \neq a)$; a summary for $Sum_{main}((a, b, c), (a', b', c'))$ corresponding to this *Inv* can be $(a' \neq b' \neq c' \neq a')$, an over-approximation of the actual program behaviour;

5.4 Informal Overview

The program shown in Fig. 5.2 has a simple loop in *main* that rotates the values in a, b, and c, in the beginning of every iteration of the *while* loop, and then conditionally makes an assignment to a. First, let us consider a version of the program in which the update to a with the return value of *bar* is replaced an assert statement, $(a \neq b)$, shown in a comment below the call to *bar*. With this modification, the program checks for the assertion $(a \neq b)$ in the loop inside *main*. The assertion is safe as a, b, and c are initialized to distinct values, and remain distinct with every execution of the loop. Note that although the conditional assignment to a invoking *foo* can violate the assertion, it is never executed as the condition is invariably false.

Verifying this program using bounded model checking, for large loop bounds, would require the loop to be unwound up to the given bound, which may not be feasible practically. This program is a good candidate for using k-induction as it can be successfully verified with three unwindings (i.e., k=3) even though the loop bound is infinite. To see this, let us go through the program and understand what is happening. The fact that a, b, and c are all distinct holds in the beginning, and is indeed a loop invariant, but we are not discovering loop invariants here. We wish to prove the assertion using an inductive argument. So, let us say that we start with arbitrary

values assigned to the variables a, b, and c, subject to the condition given in the assertion, that a and b must be distinct. Now, if go through the loop body once, a takes the value that was there in c, and b takes that one of a. This, of course, does not tell us that the new values of a and b are distinct (because we did not know if c and a were distinct in the beginning). Therefore, we assume once again that the assertion holds (i.e., we assume that c and a are distinct) and iterate through the loop once more. Now, a takes that value that was originally there in b, and b takes the one that was originally there in c. We still cannot say that a and b are distinct, because we have made only two assumptions about the arbitrarily chosen values – that a and b are not the same, and c and a are not the same. We still do not know whether the original values of b and c are same or not. Once again, therefore, we assume that the assertion holds (i.e., we assume that b and c are distinct). Together, with all these assumptions, we have discovered that all the variables have distinct values, and we can prove that no matter how many times the loop iterates after this, a and b will always be distinct. Notice that the standard (1-)induction argument would be insufficient here because the property cannot be discharged at that stage. Therefore, we strengthen the premise by unrolling the loop and attempt the proof again. Since we had to unwind the loop a total of three times, we are saying that 3-induction is sufficient here.

It is also noteworthy that if one used monolithic k-induction in which every procedure call is inlined, the size of inlined code, and hence the size of SAT instances, can grow exponentially for each unwinding. In compositional verification, we use over-approximation summaries for procedure calls and refine them, possibly with inlining as required, along with loop unwindings. In our example, using the summary *true* for *foo* is sufficient to obtain a successful k-induction proof for k=3 without the need to inline at all, resulting in smaller SAT proof instances.

However, sometimes we may need to refine one or more procedure call instances. Consider a version of the main program as shown in Fig. 5.2 which includes the call to *bar*. The assertion is now checked inside *bar*. For this program, replacing *bar* with an over-approximating summary, *true*, and unwinding the loop once will give rise to a spurious counterexample, with $(a = b)$ or $(a = c)$, during the assertion check. In order to get a successful compositional k-induction proof, we need the ability to check if a counterexample is spurious, and, if so, use the information in the counterexample trace to possibly selectively refine some of the call instances. For example, in this case we can determine only *bar* needs to be refined as the value returned by *bar* and the resulting assignment is the reason for the counterexample. The procedure call instances that are sufficient and likely candidates for refinement for a successful proof

can be determined based on an analysis of the counterexample and the code. There exists a trade-off between the overhead required for this analysis and the precision of refinement. In Section 5.5.2, we list three approaches that we have developed for selecting candidate sets of call instances for refinement. Once *bar* is refined by inlining its body, the program is verified to be safe k-induction in the next round of unwinding.

Another dimension for refinement is how to refine the selected call instances. The simplest refinement is to just inline the procedure body. A more semantic approach is to sufficiently refine the over- approximation summary. In our example, a sufficient refinement for bar can be $(a \neq b \neq c)$, or, simply, even $(a \neq b)$. In section 5.5.3, we discuss some ways for using the spurious counterexample to guide an abstract domain based summary generation tool 2LS that we use for implementing our compositional k-induction technique.

Checking if a counterexample is spurious for any given unwinding (an under-approximation of the program) can be done precisely only if every procedure call instance in the unwound code is inlined with its body or an over-approximation summary of it. Such a monolithic approach is fine as the assertion generating the counterexample is in the main program. However, if the assertion is embedded inside a procedure deep down the hierarchy, a monolithic method becomes impractical. Our compositional spurious counterexample checking method allows modular analysis of each procedure, e.g. *foo* and *bar* in Fig. 5.2, during counterexample analysis. We have implemented a number of compositional counterexample checking methods that trade-off precision to completeness (see Section 4.5). So, our compositional verification method requires a refinement step that iterates over these methods within the phase of counterexample checking for spuriousness. We describe our compositional counterexample-checking method in Section 5.5.1.

5.5 Horizontal: Interprocedural k-Induction

Horizontal, interprocedural, k-induction (Algorithm 4) differs from the monolithic one in exactly one aspect - the refinement in case of spurious counterexamples. This directly relates to the two basic building blocks on which this technique relies: the spuriousness check of counterexamples (the under-approximative right-hand side of Fig. 5.1) and the refinement of the over-approximation.

Being an interprocedural approach to k-induction, *inlining* is also a possible refinement strategy, apart from the usual refinement of monolithic approach, which is *loop*

Algorithm 4 Interprocedural k-Induction

```
1: while a proof or a counterexample is not obtained do
2:     check all properties in each procedure
3:     mark proved properties as passed
4:     for every failed property do
5:         if the failure (i.e. counterexample) is non-spurious then
6:             mark the property as failed
7:         else
8:             spuriousness check return unsat
9:             find the unsat core
10:            detect the loops and procedures which are a part of the unsat core
11:            refine – by unwinding the loops and inlining the procedure calls
```

unwinding. Besides, since the spuriousness check of counterexample is interprocedural too, this allows for "selective" loop unwinding. The next two subsections explain these.

Since the refinements that we perform eventually converge to the monolithic case, i.e. all procedures inlined and unwound to some finite k, we can therefore state that interprocedural k-induction is as complete as monolithic k-induction:

Theorem 5.5.1 (Interprocedural k-induction)**.**
i) If monolithic k-induction proves a property for some finite k then the interprocedural k-induction algorithm will prove it.
ii) If monolithic k-induction finds a counterexample after a finite k number of iterations, then the interprocedural k-induction algorithm will find it too.

Proof (sketch): We may argue by induction on the number of procedure calls.

Base case: The theorem trivially reduces to the monolithic case if there is just one procedure call.

Inductive step: Assume it holds for programs with up to n procedure calls. Consider a program with $(n + 1)$ procedure calls. Suppose we are doing interprocedural k-induction and we get a counterexample. If the counterexample is valid, then we are done. If not, while refining we would be unwinding some loops, or inlining some function calls. Since the loops can only be unwound a finite number of times, a function call would eventually get inlined as part of the refinement, giving us a program with n procedure calls. From the induction hypothesis, it follows that the inter-procedural k-induction will be able to get the same result as the monolithic k-induction.

5.5.1 Spuriousness Check of Counterexamples

Consider the example shown in Fig. 5.2 in Section 5.3. Analysing procedure *bar* in isolation gives a counterexample, which is spurious in our case as the program in Fig. 5.2 is safe.

Bounding the number of unwindings by k, a *local* counterexample for a procedure f is a solution of the formula:

$$\neg\forall \overbrace{\mathbf{x}^{in}{}_f, \mathbf{x}^{out}{}_f, \ldots}^{\text{all variables in } f} : \\ Init_f \wedge T_f^k \wedge Out_f \Longrightarrow Props_f^k \tag{5.7}$$

where T_f^k (resp. $Props_f^k$) is shorthand for

$$\bigwedge_{1\leq i\leq k} T_f(\mathbf{x}_{j,i-1}, \mathbf{x}_{j,i}) \text{ (resp. } \bigwedge_{0\leq i\leq k} Props_f(\mathbf{x}_{j,i})).$$

Instantiated on our example, we have

$$\begin{aligned} \neg\forall\ & u, v, w, u_0, v_0, w_0, u_1 : \\ & ((u = u_0 \wedge v = v_0 \wedge w = w_0) \wedge g_9 \wedge \\ & (g_{10} = g_9 \wedge \neg(u_0 = v_0)) \wedge (g_{10} \Rightarrow (u_1 = u_0))) \\ & \Longrightarrow \neg(u_0 = v_0) \end{aligned} \tag{5.8}$$

Since we are looking for a local counterexample of *bar*, we assume that the entry to *bar* is reachable, while the exit is unreachable (due to the assertion violation). In other words, the entry guard g_9 is *true* while the exit guard g_{10} is *false*. One counterexample here could be $(u_0 = v_0 = 5)$, for example. The question is now how to decide whether this counterexample is spurious or not, and to find a valid counterexample if one exists. For instance, $(u_0 = v_0 = 5)$ turns out to be spurious if we consider the whole program because it is clear from the context available in *main* that the values of u and v would never be equal when *bar* is called.

The set of *local* counterexamples found in a procedure f might contain many counterexamples that are spurious for the whole program, i.e. they are infeasible from the entry point of the program. A definite answer to this question cannot be given by looking at the local problems alone, but only by analyzing the global one.

Chapter 4 discusses refutation algorithms of varying degrees of completeness. Intuitively, the negation of the *assertion* has to be hoisted up along the error path to the entry point of the program. If the obtained weakest precondition for the violation of the assertion is not *false*, then the counterexample is feasible. Propagating up the *counterexample* itself is not sufficient to decide spuriousness as explained above.

5.5.2 Refinement Strategies

In the interprocedural setting there are two reasons for a counterexample to be spurious, namely

1. A loop invariant is too weak.
2. The calling context information for a procedure call is too weak.

The question is which of the loop invariants and calling contexts is to blame. Information to track down the culprit is obtained from a successful spuriousness check, i.e. when we have proved a counterexample to be spurious. This corresponds to an unsatisfiable formula, obtained by propagating the counterexample from the point of assertion failure to program's entry, calculating the weakest precondition of every statement along the path. The unsatisfiability of this weakest precondition formula, at any point, indicates that the counterexample cannot be propagated beyond that point, and is therefore spurious. From the (over-approximation of the) UNSAT core of this formula, we can over-approximate the set of loops and procedure calls that are involved in showing spuriousness of the counterexample.

The refinement action for a loop in this set is adding another unwinding; for procedures, it is inlining.

Based on this information, we have developed three different refinement strategies:

- *Greedy:* we always refine all elements in the set.
- *Bottom-up:* we prioritize the refinement of loops and procedures that are close to the spurious assertion violations. If a procedure contains spuriously failing assertions, all its loops are fully unwound (or it did not have loops), and all its procedure calls are inlined (or it never called any procedures), then we inline this procedure because the only reason for spuriousness is the lack of context information. In order to avoid getting stuck in unwinding deep loops we limit the number of unwindings that we do before proceeding with inlining.
- *Top-down:* we perform unwinding and inlining starting from the entry procedure downwards.

The incomplete spuriousness checks could also be used for refinement. However, they can only give directions for refinement that may turn out to be random.[1] For

[1]The over-approximation of the UNSAT core adds such randomness in the refinements.

example any loop or function call between the failing assertion and the point where the concrete spuriousness check fails (we obtain precondition *false*) is a potential candidate for refinement. Yet, it could be that the counterexample is not spurious at all, then these refinements are actually unnecessary. Hence, incomplete spuriousness checks can only help making "refinement guesses" that may not necessarily eliminate spurious counterexamples.

Example. Consider the program shown in Fig. 5.2. When analyzed separately, the procedure *bar* produces an assertion violating run (a counterexample) when $u = v$. However, this counterexample is spurious, considering one unwinding[2] of the *while* loop in *main*, since $(a_0 \neq b_0 \neq c_0 \neq a_0)$, $(a_1 = c_0 \wedge b_1 = a_0 \wedge c_1 = b_0)$, $(b_1 = c_1) \Rightarrow (a_2 = c_1)$, $(b_1 \neq c_1) \Rightarrow (a_2 = a_1)$, and $(a_2 = b_1)$ are unsatisfiable. They, in fact, form an UNSAT core. Note that the subscripts of variables a, b, and c represent the SSA form, so that each variable gets assigned exactly once (also shown in Fig. 5.2). In this case, there are no loop invariants that need strengthening. The spuriousness arises from the constraint $(a_0 \neq b_0 \neq c_0 \neq a_0)$, which is a part of the UNSAT core. Therefore, inlining gets rid of the spurious counterexample by strengthening the context.

5.5.3 Strengthen the Technique by Contexts and Invariants

The strategy used in Section 5.5.2, to refine an over-approximation of a program during verification, involved unwinding of loops for k-induction in procedures that were already inlined, and/or inlining procedures that were over-approximated, using summaries and invariants. One can use a finer refinement strategy by deferring inlining of the called procedures (or their summaries) by their body as late as required. Instead, the summaries and invariants for loops in that case are made more precise by strengthening them. The essential idea for this strengthening is to use the current over-approximated transition relation of an inlined procedure in the hierarchy to derive more precise calling context information for each called procedure instance in the inlined procedure. This calling context can then be used to narrow down the search for invariants and summaries. This may be done using 2LS [20, 96], a framework that supports automatic inference of loop invariants and function summaries, with algorithms for obtaining precise summaries and calling contexts as we discussed in the last chapter.

[2]We consider an unwinding of one because it is sufficient to uncover the reason for spuriousness in this case. Even for a larger unwinding of the *while* loop, the reason for a spurious counterexample obtained by analyzing *bar* in isolation is the same.

```
int bar(int a, b, c){
  a = *;
  while(*){
    if (b < c) assume (a > c);
    else assume (a < c);
    return a;
  }
  assert(a != b);
}
```

Figure 5.3

There are two typical situations while performing compositional k-induction where this strengthening technique is particularly useful. The first situation is depicted in Fig. 5.3, where we consider a slightly convoluted version of the procedure *bar* earlier shown in Fig. 5.2. This version has a *while* loop but is behaviourally equivalent to the original one under the calling context $(b \neq c)$. If we are performing k-induction over the *while* loop in *bar* after it has been inlined, then instead of starting from an arbitrary state in the inductive step, one can refine the arbitrariness to conform to the calling context information available at the call instance of *bar*. The precise calling context for the calling instance of *bar* in *main*, of Fig. 5.2, is $(b \neq c)$. With this calling context information passed, 2LS will be able to generate the invariant $(a \neq b \wedge a \neq c)$ for the loop in *bar* and the same formula as summary for *bar*. This refined summary can then be used to make the k-induction proof of the main loop to succeed without inlining *bar*.

The second situation is illustrated by the *while* loop in *main* in Fig. 5.2, when there is a call to a function from inside the loop. Suppose, for instance, that the initial condition of a, b, and c being all distinct is not there. I.e., any of these variables may be equal to any other variable, or even to both the other variables, before entering the *while* loop. Now, if we analyze the loop code with the over-approximated summary *true* for *foo*, we would get the context $(b = c)$. In that case, we would generate another summary of *foo* by taking this context into account, and thereby obtain a more precise summary for *foo* to be used in the next iteration.

There are two steps essential in strengthening summaries and invariants, namely (i) a *forward calling context* computing step that computes the calling context by hierarchically traversing the call graph, and (ii) a *backward strengthening* step in which the invariants and summaries are strengthened using the calling contexts generated

in the first step. The calling context derivation and strengthening of summaries and invariants is done by means of generating witnesses to the existentially quantified formulas, as per the definitions below.

Definition 5.5.2. *A forward calling context $CallCtx_{h_i}$ for h_i in procedure f in calling context $CallCtx_f$ is a satisfiability witness of the following formula:*

$$\begin{aligned}\exists CallCtx_{h_i}, Inv_f : \ & \forall \mathbf{x}^{in}, \mathbf{x}, \mathbf{x}', \mathbf{x}^{out}, {\mathbf{x}^{p\_in}}_i, {\mathbf{x}^{p\_out}}_i : \\ & CallCtx_{h_i}(\mathbf{x}^{in}, \mathbf{x}^{out}) \wedge Sums_f \implies (Init_f(\mathbf{x}^{in}, \mathbf{x}) \implies Inv_f(\mathbf{x})) \\ & \wedge (Inv_f(\mathbf{x}) \wedge T_f(\mathbf{x}, \mathbf{x}') \implies \\ & \quad Inv_f(\mathbf{x}') \wedge (g_{h_i} \implies CallCtx_{h_i}({\mathbf{x}^{p\_in}}_i, {\mathbf{x}^{p\_out}}_i))\end{aligned}$$

with

$$Sums_f = \bigwedge_{calls\ h_j\ in\ f} (g_{h_j} \implies Sums[h]({\mathbf{x}^{p\_in}}_j, {\mathbf{x}^{p\_out}}_j))$$

where g_{h_j} is the guard condition of procedure call h_j in f, and $Sums[h]$ is the currently available summary for h.

Definition 5.5.3. *A forward summary Sum_f and invariants Inv_f for procedure f in calling context $CallCtx_f$ are satisfiability witnesses of the following formula:*

$$\begin{aligned}\exists Sum_f, Inv_f : \ & \forall \mathbf{x}^{in}, \mathbf{x}, \mathbf{x}', \mathbf{x}'', \mathbf{x}^{out} : \\ & CallCtx_f(\mathbf{x}^{in}, \mathbf{x}^{out}) \wedge Sums_f \implies \\ & \quad (Init_f(\mathbf{x}^{in}, \mathbf{x}) \wedge Inv_f(\mathbf{x}'') \wedge Out_f(\mathbf{x}'', \mathbf{x}^{out}) \\ & \quad\quad \implies Inv_f(\mathbf{x}) \wedge Sum_f(\mathbf{x}^{in}, \mathbf{x}^{out})) \\ & \wedge (Inv_f(\mathbf{x}) \wedge T_f(\mathbf{x}, \mathbf{x}') \implies Inv_f(\mathbf{x}'))\end{aligned}$$

These formulae are similar to those in Def 5.5.2, except that now we are not looking for a predicate over input/output variables of the caller, but over arguments/return values of a callee. Since Def. 5.5.2 and Def. 5.5.3 are interdependent, we can compute them iteratively until a fixed point is reached in order to improve the precision of calling contexts, invariants and summaries. The previous chapter illustrates a forward summary algorithm, Alg. 3, showing how this may be done for programs with a fixed unwinding of loops.

5.6 Concluding Remarks

In this chapter we proposed an approach to interprocedural k-induction, based on interprocedural spuriousness check of counterexamples explained in the last chapter. We formulated this within a generic refinement framework - one that selectively unwinds loops and inlines procedures. The interprocedural k-induction is as complete as the monolithic case, since the refinement eventually converges to all procedures

inlined and unwound up to the given k. We also looked at deferring inlining of the over-approximated procedures by using summaries. The summaries may be imprecise to begin with, but can be strengthened through the calling context information. This chapter gives a theoretical illustration of how *horizontal decompositions*, around which the compositional safety refutation techniques were devised in Chapter 4, can also help us obtain safety proofs compositionally. The implementation and experimental evaluation of this work are not a part of this thesis.

5.6.1 Notes

While this chapter uses k-induction to complete the compositional verification framework with unwinding and inlining as refinement, one might equally well couple a different abstraction-refinement strategy with the compositional refutation techniques. The strategy may depend on how the system has been split into components. The splitting criteria may even be semantic, not just syntactic. As we conclude in the next chapter, we also discuss some prospects of our work.

Chapter 6

Conclusion

In this work, we looked at three important aspects integral to Statechart-*like* systems, namely *concurrency*, *cyclicity* and *size*, and came up with techniques to overcome their effect on scalability of software verifiers. In particular, we were interested in finding an answer to the following questions:

1. For synchronous reactive systems modeled as STATEMATE Statecharts, is it possible to analyze synchronous concurrency explicitly, rather than encoding it as part of the system's implementation?

2. Do loop accelerators enable existing program analysis algorithms to discover loop invariants more reliably and more efficiently?

3. Is it possible to generate counterexamples in a modular way to speed up safety refutation?

4. Is there a compositional approach to k-induction, in an abstraction-refinement framework that allows a component-wise refinement instead of a monolithic one?

We list down, and then discuss, the key contributions made by this thesis, in an attempt to answer these four questions.

- We proposed a technique tailored for verifying synchronous reactive systems, as an extension of the LAwI algorithm implemented in IMPARA. We also described an implementation of our technique into a tool called SYMPARA.

- We quantified the benefit of acceleration, as a source-level transformation, for checking safety properties. We reported the results of a comprehensive comparison over a number of benchmarks, showing that the combination of acceleration and a safety checker outperforms existing tools and techniques, for both safe and unsafe benchmarks.

- We formalised the problem space of safety refutation in horizontal decompositions and characterised the compositional completeness guarantees of various algorithmic approaches. We also described three refutation approaches with different degrees of completeness, and compared their completeness and efficiency through experiments.

- We formulated verification by k-induction within a generic refinement algorithm framework, and proposed a horizontal, i.e. interprocedural k-induction approach based on an interprocedural counterexample spuriousness check and a selective refinement of loop unwindings and procedure inlining.

Analyzing synchronous reactive systems, under the semantics that we were looking at, does not require one to address *concurrency* in the most general setting. However, existing verifiers of such systems often flatten them into a global transition system, to be able to apply off-the-shelf verification methods. This monolithic approach fails to exploit the lock-step execution of the processes, severely limiting scalability.

We presented a novel formal verification technique that analyses synchronous concurrency explicitly rather than encoding it. We proposed a variant of Lazy Abstraction with Interpolants (LAwI), a technique successfully used in software verification, and tailored it to synchronous reactive concurrency. We were able to exploit the synchronous communication structure by fixing an execution schedule, thus circumventing the exponential blow-up of state space caused by simulating synchronous behaviour by means of interleavings. We also implemented this technique in a tool called Sympara, and shared experimental results on realistic examples showing that Sympara outperformed Bounded Model Checking with k-induction, and a LAwI-based verifier for multi-threaded (asynchronous) software, by an order of magnitude.

Addressing the issue of *cyclicity*, we quantified the benefits that acceleration offers for checking safety properties of programs. Acceleration is a technique for summarising loops by computing a closed-form representation of the loop behaviour. The closed form can be turned into an *accelerator*, which is a code snippet that skips over intermediate states of the loop to the end of the loop in a single step.

We experimentally evaluated whether loop accelerators enable *existing* program analysis algorithm to discover loop invariants more reliably and more efficiently. This work was the first comprehensive study on the synergies between acceleration and invariant generation. We reported our experience with a collection of safe and unsafe programs drawn from the Software Verification Competition and the literature.

While attending to the aspect of *size*, we chose to deal with the problem of refutation prior to that of proofs. In fact, our approach to finding safety proofs employs our refutation strategies to solve a sub-problem. We look for a modular approach as even the most successful techniques for refuting safety properties (finding counterexamples by bounded model checking) exceed the limit of resources available, on most model checking instances for large programs. Generating such counterexamples in a modular way could speed up refutation but the counterexamples are inherently non-compositional in nature, making it a challenging task. We formalised a space of property-guided compositional refutation techniques, discussed their properties with respect to efficiency and completeness, and presented an experimental evaluation of the techniques.

For proofs, we looked at k-induction – one of the most popular techniques for proving safety. With the similar motivation that performing k-induction in a modular way could speed up verification, we proposed an interprocedural approach to modular k-induction, as an instance of a more general refinement approach to program verification.

Additionally, since all our implementations have been done on the CPROVER framework, if there is practical interest, it would be possible to investigate how a combination of these techniques fare.

The techniques developed, or proposed, as part of this work are essentially state-space reduction techniques. This is not surprising – state-space explosion lies at the core of automated software verification. So any technique that addresses scalability issues, must reduce the state-space available for exploration. Our approach for verifying synchronous reactive systems achieves this reduction through concurrent SSAs and early elimination of infeasible paths. For loop acceleration, this is achieved by skipping over intermediate states of a loop. Compositional techniques, described in the last two chapters, accomplish this by analyzing the modules in isolation, as far as possible, and composing the results in the end. Overall, this piece of work is aimed at scalable verification, and the techniques have been developed keeping in mind the crucial aspects of Statechart-*like* systems.

6.1 Prospects

The work in this thesis has led to some appealing ideas that would be worthwhile to explore.

- In general, it would be interesting to see if an IMPACT like algorithm can benefit from an external module that supplies invariants. Such a module may also

accelerate invariant generation, beyond loop acceleration. Our work has laid the platform needed to experiment with this, and to quantify its benefits.

- It would be rewarding to implement the techniques that we have theoretically illustrated, in Chapter 5, for obtaining compositional safety proofs using k-induction.

- For the compositional approaches to safety refutation and proofs, it would be beneficial to explore different ways of procedure summarization. These summaries may be of different degrees of precision, and may or may not be property-directed. The modular framework described in this thesis allows and assists future work in this direction.

The work in this thesis has also opened up new directions that would be interesting to explore. For instance, while this thesis studies the modularity of implementations based on the syntax (i.e., we split a system along procedures boundaries), it would be gainful to explore if there are *semantic boundaries* that can be used for decomposing a system into modules, such that compositional verification becomes easier. This semantic split could even be motivated by the property to be verified. There seem to be several ways in which this can be studied and, perhaps, this would be worthy of an entire doctoral thesis in itself.

A short-term project, although a very useful one, would be to extend our work on compositional refutation to generate integration test cases from unit tests. It might be valuable to see if, given a set of unit tests (concrete summaries) for a subset of functions, it is possible to obtain a set of integration tests (counterexamples) starting from program entry point that has same coverage as unit tests (or show that some of these tests are infeasible when starting from the entry point).

Bibliography

[1] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *In CAV'98. LNCS 1427*, pages 305–318. Springer-Verlag, 1998.

[2] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for interprocedural verification. In *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2012.

[3] Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. UFO: Verification with interpolants and abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *LNCS*, pages 637–640. Springer, 2013.

[4] Aws Albarghouthi and Kenneth L. McMillan. Beautiful interpolants. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 313–329. Springer-Verlag, Berlin, Heidelberg, 2013.

[5] R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In E.Allen Emerson and AravindaPrasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 280–295. Springer Berlin Heidelberg, 2000.

[6] Rajeev Alur, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Automating modular verification. In *Concurrency Theory*, pages 82–97, 1999.

[7] Rajeev Alur, Michael McDougall, and Zijiang Yang. Exploiting behavioral hierarchy for efficient model checking. In Ed Brinksma and KimGuldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 338–342. Springer Berlin Heidelberg, 2002.

[8] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[9] Sébastien Bardin, Alain Finkel, and Jérôme Leroux. Faster acceleration of counter automata in practice. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 576–590. Springer, 2004.

[10] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation, 1992.

[11] Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2015.

[12] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 300–309. ACM, 2007.

[13] Dirk Beyer and M.Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.

[14] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. *CSIsat: Interpolation for LA+EUF*, pages 304–308. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[15] Purandar Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions. *CoRR*, cs.SE/0407038, 2004.

[16] Tom Bienmüller, Werner Damm, and Hartmut Wittke. The Statemate verification environment – making it real. In *Computer Aided Verification (CAV)*, pages 561–567. Springer, 2000.

[17] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 831–848. Springer International Publishing, 2014.

[18] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. Collection des publications. Université de Liège, Faculté des sciences appliquées, 1999.

[19] Marius Bozga, Radu Iosif, and Filip Konecný. Fast acceleration of ultimately periodic relations. In *Computer Aided Verification*, volume 6174 of *LNCS*, pages 227–242. Springer, 2010.

[20] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety Verification and Refutation by k-Invariants and k-Induction. In *Static Analysis Symposium*, volume 9291 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2015.

[21] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety verification and refutation by k-invariants and k-induction (extended version). Technical report, University of Oxford, UK, 2015. `http://arxiv.org/abs/1506.05671`.

[22] Jörg Brauer, Andy King, and Jael Kriener. Existential quantification as incremental SAT. In *Computer-Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 191–207. Springer, 2011.

[23] Jörg Brauer and Axel Simon. Inferring definite counterexamples through under-approximation. In *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2012.

[24] RobertK. Brayton, GaryD. Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, RajeevK. Ranjan, Shaker Sarwary, ThomasR. Staple, Gitanjali Swamy, and Tiziano Villa. Vis: A system for verification and synthesis. In Rajeev Alur and ThomasA. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer Berlin Heidelberg, 1996.

[25] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[26] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[27] William Chan, Richard J. Anderson, Paul Beame, David Notkin, David H. Jones, and William E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Trans. Softw. Eng.*, 27(2):170–190, February 2001.

[28] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. Synthesising interprocedural bit-precise termination proofs. In *Automated Software Engineering*, pages 53–64. ACM, 2015.

[29] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. *The MathSAT5 SMT Solver*, pages 93–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[30] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Verifying SystemC: A software model checking approach. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 51–60, 2010.

[31] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[32] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[33] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[34] Edmund M. Clarke and Wolfgang Heinle. Modular translation of statecharts to SMV. Technical Report CMU-CS-00-XXX, School of Computer Science, Carnegie Mellon University, April 2000.

[35] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *Logic in Computer Science*, pages 353–362. IEEE Computer Society, 1989.

[36] E.M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[37] The GNU Compiler Collection. `https://gcc.gnu.org/`.

[38] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *Computer-Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 328–340. Springer, 2008.

[39] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[40] 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 Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[41] Priyanka Darke, Bharti Chimdyalwar, R. Venkatesh, Ulka Shrotri, and Ravindra Metta. Over-approximating loops to prove properties using bounded model checking. In *Design, Automation & Test in Europe (DATE)*, pages 1407–1412. EDA Consortium, 2015.

[42] Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. Danger invariants. In *Formal Methods*, volume 9995 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2016.

[43] Giorgio Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Form. Methods Syst. Des.*, 23(3):257–301, November 2003.

[44] A. Donaldson, L. Haller, D. Kroening, and Philipp Rümmer. Software Verification Using k-Induction. In *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2011.

[45] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronical Notes in Theoretical Computer Science*, 89:4:543–560, 2003.

[46] Alain Finkel and Jérôme Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Foundations of Software Technology and Theoretical Computer Science (FST TCS)*, volume 2556 of *LNCS*, pages 145–156. Springer, 2002.

[47] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[48] Robert W. Floyd. Assigning meanings to programs. In *Proc. Sympos. Appl. Math., Vol. XIX*, pages 19–32. Amer. Math. Soc., Providence, R.I., 1967.

[49] A C/C++ front-end for Verification. `http://www.cprover.org/goto-cc/`.

[50] Pierre-Loïc Garoche, Temesghen Kahsai, and Cesare Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2013.

[51] Thierry Gautier, Paul Le Guernic, and Löic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 257–277, London, UK, UK, 1987. Springer-Verlag.

[52] Thomas M. Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2007.

[53] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[54] Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 112–128, Berlin, Heidelberg, 2011. Springer-Verlag.

[55] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[56] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Dagger Benchmarks Suite. `http://www.cfdvs.iitb.ac.in/~bhargav/dagger.php`, 2014.

[57] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Programming Language Design and Implementation*, pages 281–292. ACM, 2008.

[58] Ashutosh Gupta and Andrey Rybalchenko. InvGen Benchmarks Suite. `http://pub.ist.ac.at/~agupta/invgen/`, 2014.

[59] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *FMCAD*, pages 1–9. IEEE Computer Society, 2008.

[60] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[61] David Harel and Amnon Naamad. The Statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[62] Mark Harman and Robert M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[63] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *Static Analysis (SAS)*, volume 5673 of *LNCS*, pages 69–85. Springer, 2009.

[64] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Termination analysis by learning terminating programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 797–813. Springer, 2014.

[65] Hossein Hojjat, Radu Iosif, Filip Konecný, Viktor Kuncak, and Philipp Rümmer. Accelerating interpolants. In *Automated Technology for Verification and Analysis (ATVA)*, LNCS, pages 187–202. Springer, 2012.

[66] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[67] JavaBDD. `http://javabdd.sourceforge.net/`.

[68] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract acceleration of general linear loops. In *Principles of Programming Languages (POPL)*, pages 529–540. ACM, 2014.

[69] Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

[70] Temesghen Kahsai, Yeting Ge, and Cesare Tinelli. Instantiation-based invariant discovery. In *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2011.

[71] Anvesh Komuravelli, Nikolaj Bjørner, Arie Gurfinkel, and Kenneth L. McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *Formal Methods in Computer-Aided Design*, pages 89–96. IEEE Computer Society, 2015.

[72] D. Kroening, M. Lewis, and G. Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *Computer-Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 381–396. Springer, 2013.

[73] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *Computer Aided Verification (CAV)*, pages 381–396. Springer, 2013.

[74] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Proving safety with trace automata and bounded model checking. In *Formal Methods (FM)*, volume 9109 of *LNCS*. Springer, 2015.

[75] Akash Lal and Shaz Qadeer. *Reachability Modulo Theories*, pages 23–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[76] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the Spin model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[77] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*,

volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[78] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with SMT solvers. In *Principles of Programming Languages*, pages 607–618. ACM, 2014.

[79] Kumar Madhukar, Peter Schrammel, and Mandayam K. Srivas. Compositional safety refutation techniques. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 164–183. Springer, 2017.

[80] Kumar Madhukar, Mandayam K. Srivas, Björn Wachter, Daniel Kroening, and Ravindra Metta. Verifying synchronous reactive systems using lazy abstraction. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 1571–1574. ACM, 2015.

[81] Kumar Madhukar, Björn Wachter, Daniel Kroening, Matt Lewis, and Mandayam K. Srivas. Accelerating invariant generation. In Roope Kaivola and Thomas Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 105–111. IEEE, 2015.

[82] K. L. Mcmillan. Lazy abstraction with interpolants. In *Computer-Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[83] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification (CAV)*, pages 123–136. Springer, 2006.

[84] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing statecharts in promela/spin. In *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, WIFT '98, pages 90–, Washington, DC, USA, 1998. IEEE Computer Society.

[85] Antoine Miné. Inferring sufficient conditions with backward polyhedral underapproximations. *Electr. Notes Theor. Comput. Sci.*, 287:89–100, 2012.

[86] MiniSAT. `http://minisat.se/MiniSat.html`.

[87] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Softw. Eng.*, 7(4):417–426, July 1981.

[88] David Monniaux and Peter Schrammel. Speeding up logico-numerical strategy iteration. In *Static Analysis Symposium*, volume 8723 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2014.

[89] Kedar S. Namjoshi and Richard J. Trefler. On the completeness of compositional reasoning methods. *ACM Trans. Comput. Log.*, 11(3), 2010.

[90] A. Pnueli. Logics and models of concurrent systems. chapter In Transition from Global to Modular Temporal Reasoning About Programs, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[91] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2004.

[92] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[93] A.W. Roscoe and Z. Wu. Verifying Statemate statecharts using CSP and FDR. In *Formal Methods and Software Engineering*, volume 4260 of *LNCS*, pages 324–341. Springer, 2006.

[94] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2006.

[95] Peter Schrammel. Challenges in decomposing encodings of verification problems. In *Horn Clauses for Verification and Synthesis*, Electronic Proceedings in Theoretical Computer Science, pages 29–32, 2016.

[96] Peter Schrammel and Daniel Kroening. 2LS for Program Analysis - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *Lecture Notes in Computer Science*, pages 905–907. Springer, 2016.

[97] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Successful use of incremental BMC in the automotive industry. In *Formal Methods for Industrial Critical Systems*, volume 9128 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2015.

[98] Roberto Sebastiani and Silvia Tomasi. Optimization in SMT with LA cost functions. In *International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 484–498. Springer, 2012.

[99] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-based function summaries in bounded model checking. In *Haifa Verification Conference*, volume 7261 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2011.

[100] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[101] SPARK. `http://www.spark-2014.org/`, 2014.

[102] Aditya V. Thakur and Thomas W. Reps. A method for symbolic computation of abstract operations. In *Computer-Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 174–192. Springer, 2012.

[103] Tamás Tóth and András Vörös and István Majzik. K-induction based verification of real-time safety critical systems. In *DepCoS-RELCOMEX*, 2013.

[104] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with Impact. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 210–217, 2013.

[105] Qianchuan Zhao and B.H. Krogh. Formal verification of statecharts using finite-state model checkers. *Control Systems Technology, IEEE Transactions on*, 14(5):943–950, Sept 2006.

[106] Qianchuan Zhao and Bruce H. Krogh. Formal verification of statecharts using finite-state model checkers. In *In American Control Conference, 2001. Proceedings of the 2001*, pages 313–318. IEEE, 2001.