Compositional Verification with Abstraction, Learning, and SAT Solving

Anvesh Komuravelli

CMU-CS-15-102
May 2015

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Edmund M. Clarke, Chair
Jonathan Aldrich
Frank Pfenning
Aarti Gupta, Princeton University
Kenneth L. McMillan, Microsoft Research
Corina S. Pasareanu, NASA Ames/CMU Silicon Valley

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

This research was sponsored by the Air Force Office of Scientific Research under grant number FA9550-06-1-0312, Microelectronics Advanced Research Corporation (MARCO) under grant number 2009-DT-2049, Semiconductor Research Corporation under grant number 2008-TJ-1860, National Science Foundation under grant number CNS-0926181, Office of Naval Research under grant number N000141010188, Air Force Research Laboratory under grant number FA9550-12-1-0146, Defense Advanced Research Projects Agency under grant number F8721-05-C-0003, and the Portuguese Science Technology Foundation.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. Government, or any other entity.

Keywords: model checking, verification, proof, counterexample, satisfiability, SAT, SMT, oracle, abstraction, refinement, approximation, logic, quantifier elimination, decision procedure, program, software, loops, recursion, procedures, Z3, induction, safety, modular, compositional, invariant, interpolation, cegar, proof-based abstraction, symbolic, parallel, probability, transition system, simulation, active learning, undecidable

To my parents

To the eternal mystery of the human mind

To love and music

To nature

Abstract

Compositional reasoning is an approach for scaling model checking to complex computer systems, where a given property of a system is decomposed into properties of small parts of the system. The key difficulty with compositional reasoning is in automatically coming up with sufficient decompositions of global properties into local properties. This thesis develops efficient compositional algorithms for safety of (a) sequential recursive programs, using solvers for SAT and SAT modulo theories (SMT), and (b) parallel, finite-state probabilistic systems. These algorithms result in significant improvements over the state-of-the-art, both in theory and in practice.

For SAT-based verification of sequential programs, monolithic techniques based on *Bounded Model Checking* (BMC) iteratively check satisfiability of formulas whose size can grow exponentially in the input size of the program. While safety can be decided in time polynomial in the number of states, existing SAT-based algorithms do not have such guarantees. We develop a compositional SAT-based algorithm that maintains and utilizes *under-* and *over-approximations* of the behavior of procedures. While addressing the above complexity problem, the algorithm also extends to realistic programs that involve arithmetic operations using oracles for SMT.

In order to improve practical convegence of the iterative approach for SMT-based verification, we also develop a new mechanism for *automatic abstraction refinement* of the input program. This combines ideas from *Proof Based Abstraction* (PBA) and *CounterExample Guided Abstraction Refinement* (CEGAR) in the literature.

We describe SPACER (*Software Proof-based Abstraction with CounterExample-based Refinement*), a tool that implements the above algorithms, using which we show significant advantages on realistic benchmarks.

For probabilistic transition systems with multiple parallel components, the number of states of a system can grow exponentially in the number of components (the well-known *state-space explosion* problem). For these systems, we develop the first compositional algorithms for checking simulation conformance. We follow an assume-guarantee style reasoning and establish theoretical bounds on the learnability of an intermediate assumption of the least number of states from positive and negative examples. We also develop a practical algorithm based on abstraction refinement. Using a Java implementation of the latter, we show practical advantage over monolithic verification.

Acknowledgments

Working with Ed Clarke has been a once-in-a-lifetime experience. Ed has not only taught me what it takes to do great research, but also given me the courage to ask basic questions in research. Thank you Ed for spending your valuable time listening to my half-baked ideas during countless number of meetings! The presentation in this document has remarkably improved over time, thanks to the careful and detailed reading by Ed. I cannot thank Ed without also thanking his better half, Martha, who was always very supportive. Thanks Martha for all the wonderful parties at your house!

This thesis would have been impossible without the guidance of Arie Gurfinkel. At a point when I was not even interested in software verification because I thought it was "over-crowded", I met Arie who got me interested in it and patiently taught me the subject ground-up. More importantly, he was always there to help me out with research! Thanks to Sagar Chaki for all the positive feedback that helped shape much of the work in this thesis.

I was fortunate to have worked with Corina Păsăreanu who kindly agreed to mentor me for a summer in 2012. Starting with a really hard problem that I had spent more than a semester on making very little progress, Corina helped me find another hard problem, solve it, and write two papers about it, forming the basis of a significant chunk of this thesis. I must also thank Jonathan Aldrich, Aarti Gupta, Ken McMillan, and Frank Pfenning for serving on my thesis committee and giving me invaluable feedback. I thank Christel Baier, Nikolaj Bjørner, Rohit Chadha, Lu Feng, Holger Hermanns, Marta Kwiatkowska, Joel Ouaknine, David Parker, Cesare Tinelli, Frits Vaandrager, Mahesh Viswanathan, James Worrell, and Lijun Zhang for generously answering questions related to this research at various points or giving valuable feedback on early drafts of papers.

Thanks are also due to Sicun Gao, Samir Sapra, and Nishant Sinha for all the fruitful discussions and for always being there during stressful times! Debbie Cavlovich and Charlotte Yano were two indispensable people who made my life at CMU so much easier!

Music has been a significant part of my non-academic life at CMU. I thank the Indian Graduate Student Association and the fellow students of the Indian music team for giving me a platform to take rebirth in music, after a decade of zero activity. Ankit Sharma has been a constant source of encouragement, be it for restarting singing or accompanying me to meet with a famous Indian singer at her CMU visit when I was too shy to do so (I ended up chatting with her for more than an hour). I feel lucky to have intersected with Ashique KhudaBukhsh who has been a mentor and

a source of immense inspiration. I thank Kaushik Lakshminarayanan for generously agreeing to teach me Carnatic Music *for free*. I owe much of my growth in music over the years at CMU, to whatever extent it is worth a mention, to Ashique and Kaushik. Thanks to NEC Labs for keeping an electronic keyboard in their lounge during my internship where I first got interested in the instrument and thanks to Carla LaRocca at CMU for patiently teaching me proper piano.

Thanks go to Ankit again for organizing many fun dinners and to Anuj Kumar, Mehdi Samadi, and their numerous roommates, for making sure fun does not leave the table. I will miss the awesome folks on the 7th floor of GHC and all the intellectual lunch discussions we used to have about all the worldly affairs. Thank you Athula Balachandran, Jeremiah Blocki, David Henriques, Debbie Katz, Akshay Krishnamurthy, Sarah Loos, João Martins, Dana and Yair Movshovitz-Attias, Yuzi Nakamura, Aaditya Ramdas, Gabe Weisz, and John Wright! Special thanks to Ankit, Carol, and Yuzi for spending many hours giving useful feedback on some of my presentations. I also thank my office-mates for the occasional engaging discussions and more importantly for showing up almost never, leaving the huge office to myself. I should also thank Zack Coker, Ashique, David Kurokawa, Shayak Sen, Sahil Singla, and John Wright for all the fun times playing table-tennis (ping-pong).

And finally, the people who have given me constant emotional support, who were there during ups and downs, who shared my joys and sorrows: my parents Indira and Madhukar, and brother Rakesh – I can't thank you enough.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Model checking [39] is an automatic technique for verifying correctness properties of computer systems. Since its invention in the 1980's, numerous approaches have been proposed for scaling model checking to complex and real-world systems. We are interested in the approach of *compositional reasoning*, where the basic idea is to decompose a given property of a system into properties of small parts of the system. If the local properties together imply the overall property of the system, it suffices to check each of the local properties. Such an approach can be very efficient in practice as the whole system can be exponentially more complex than the individual parts combined. For example, the size of the state-space of a reactive system composed of multiple components running in parallel can grow exponentially in the number of components and this phenomenon is well-known as the *state-space explosion*. Several frameworks have been developed for compositional reasoning of such reactive systems (e.g., [35, 98]). In the context of program verification, Hoare logic [73], in

particular the *Rule of Composition* and the *Rule of Recursion*, can also be seen as a compositional framework for checking partial correctness triples. Here, a local property corresponds to a Hoare triple for an individual statement or a procedure. With Hoare logic, it suffices to find one generic local property per procedure that can be *adapted* to analyze every call to the procedure [37], despite the possibility of exponentially many such calls in an execution of the program it is part of. However, the main challenge in compositional reasoning is to automatically come up with a sufficient decomposition of an overall property of the system into local properties. In this thesis, we develop several efficient algorithms for automatically discovering such decompositions to check *safety* of *recursive programs* and *probabilistic systems with multiple, parallel components*.

1.1 SMT-Based Software Model Checking

The first step in software model checking is to identify the logical systems used to model the various program operations and express predicates describing the program's behavior. In this thesis, we use first-order languages for these purposes. In particular, we are interested in model checking techniques that are based on checking satisfiability of logical formulas in the languages.

The introduction of Boolean satisfiability (SAT) solvers in model checking has revolutionized the field and SAT-based algorithms are some of the best we have today. The idea of using a SAT solver for model checking was first introduced by Biere et al. [22] using a technique called *Bounded Model Checking* (BMC). BMC was

proposed for the verification of a (symbolically represented) Kripke structure against temporal logic specifications for a given bound on the length of a counterexample.

In order to extend SAT-based methods to software model checking, one needs an oracle for satisfiability of formulas in the underlying first-order language. However, as allowing arbitrary interpretations of the various program operations is undesirable, one typically utilizes a first-order theory of sentences to characterize the intended interpretation of the operations. For example, Presburger Arithmetic characterizes linear arithmetic over integers. Thus, we need an oracle for satisfiability *modulo theories* (SMT). Then, to obtain a BMC procedure for safety in sequential programs, the bound is typically on the number of loop iterations (e.g., see [40]) and on the depth of recursion[1] (e.g., see [7, 92]), which implicitly bounds the length of a counterexample. In other words, the bound b corresponds to all executions that make at most b nested calls and that use at most b iterations of any given loop. We use the term *bounded safety* to refer to the problem of checking safety for a given bound (in the above sense; see Chapters 2, 3 for details). To prove safety, we use Hoare triples for *partial correctness* specifications. Here, given assertions φ, ψ in the underlying first-order language and a statement τ, a Hoare triple $\{\varphi\}\ \tau\ \{\psi\}$ specifies that whenever τ is executed from a state satisfying φ, it will either fail to terminate or end in a state satisfying ψ. There exist sound and complete (in the sense of Cook [41]) Hoare proof systems for while programs [73] and procedural programs [36, 37].

[1] In general, a procedure can call other procedures, perhaps in a mutually recursive way, in which case bounding the *call-stack* leads to a more systematic approach. See Chapter 2 for details.

```
Main () {
  bool b := nd();
  Level<1> (b);
  Level<1> (b);
  assert (b);
}
```

```
Level<i> (bool b) {
  if (!b) {
      Level<i+1> (b);
      Level<i+1> (b);
  }
  b := !b;
}
```

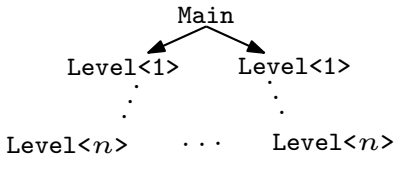


Figure 1.1: A Boolean program with exponential unwinding size.

1.1.1 Efficient Bounded Safety

Existing BMC algorithms for bounded safety create SMT problems that may grow exponentially with the bound on the recursion depth due to the tree-like unrolling of the call-graph. For example, Fig. 1.1 shows a program with Boolean variables (adapted from [19]) and finitely many `Level<i>` procedures. Here, `nd` is a routine that returns an unknown Boolean value.[2] For a bound n on the number of such procedures, assuming that procedure calls in the body of `Level<`n`>` are replaced by `noop`s, the figure also shows its tree-like unrolling which grows exponentially in n. With one Boolean parameter per procedure, note that the number of program states is linear in n, where a state corresponds to a valuation of the program counter and the variables in scope. Therefore, many of present-day BMC-based model checking algorithms, e.g., Whale [7], HSF [63], Ultimate Automizer [68, 69], Duality [92], are at least worst-case exponential in the number of states for Boolean programs. However, note that the operational semantics of a Boolean program (with the crucial assumption that procedures are disallowed as parameters [36]) can be defined in terms of a pushdown automaton where the push and pop operations on the stack correspond

[2]In other words, assume that the behavior of `nd` is unknown. So, for the purpose of verification, `nd` effectively returns either `true` or `false` non-deterministically.

to procedure calls and returns, and the accepting states denote the safe program states. This reduces safety in Boolean programs to state-reachability in pushdown automata and there exist polynomial-time (cubic) algorithms for the latter that are not SAT-based [11, 19, 101].

On the other hand, the algorithm GPDR [74] follows the approach of IC3 [25] by solving BMC incrementally without unrolling the call-graph. For some configurations (e.g., explicit-state reasoning), GPDR is worst-case polynomial for Boolean Programs. However, it gets more challenging when the program operations and formulas are in a first-order language. In this case, GPDR might even fail to find a counterexample despite the presence of an SMT oracle, unlike the guarantee given by other BMC-based algorithms mentioned above (see Appendix 2.A for an example).

To address the aforementioned problems, we present a new SMT-based algorithm for analyzing the program compositionally. That is, we iteratively check safety properties of individual procedures and infer *approximations* about their input-output behavior, by making use of the previously inferred approximations of the procedures being called. Our main insight is to utilize not only over-approximations of procedure behaviors, as in existing algorithms, but also their *under-approximations*.

For Boolean Programs, this results in a terminating algorithm for safety, without any bound, and has a polynomial time complexity (see Chapter 2). Moreover, in general, assuming an SMT oracle for the first-order language of the assertions and the program operations, we show that our compositional algorithm terminates for bounded safety. To the best of our knowledge, this is the first SMT-based algorithm which such guarantees. Details are discussed in Chapter 2.

1.1.2 Better Proofs of Bounded Safety

While bounded safety of programs is reducible to SMT (via BMC), *unbounded* safety of programs is undecidable in general.[3] Many of present-day algorithms for SMT-based model checking follow an iterative approach by checking bounded safety for increasing values of the bound on the loop iterations and the recursion depth. Each iteration of these algorithms corresponds to checking whether there is a counterexample to safety for the given bound on the executions by means of an SMT solver (using BMC). If the SMT solver returns *sat*, a counterexample to safety is obtained using a satisfying assignment. On the other hand, if the SMT solver returns *unsat*, the algorithms utilize techniques for Craig Interpolation [43] to obtain assertions that *over-approximate* the reachable states at various program locations, sufficient to show bounded safety (e.g., see [89]). These assertions constitute a Hoare-style proof of bounded safety. Now, if these assertions are also *invariant* for the program, i.e., hold for every execution of the program,[4] the algorithms terminate. However, it can be very challenging in practice for the assertions obtained from the proofs of bounded safety to be invariant, given the undecidability of safety.

To obtain better proofs of bounded safety, we describe an algorithm for *automatic abstraction refinement* of the input program, i.e., the algorithm tries to infer conservative over-approximations of the transition relation sufficient to obtain program invariants. The key intuition is that conservative abstractions help us infer better approximations of reachable states when proving bounded safety, that are easier to

[3]This follows from the undecidability of safety in a two-counter machine.
[4]This can be checked using an inductive argument.

generalize to program invariants. Moreover, as proofs, in general, do not depend on all the details of a program, we can also obtain abstractions by hiding the details irrelevant for proofs of bounded safety (via *Proof-based Abstraction* [66, 91]). Such an abstraction is used to obtain a proof of bounded safety for a bigger bound in the next iteration. Thus, our algorithm has a tight connection between proofs and abstractions. When the abstractions are too coarse, we use spurious abstract counterexamples to refine them (via *CounterExample-Guided Abstraction Refinement* [38]). This approach is also compositional as it tries to obtain an abstraction of the transition relation, i.e., the *data component* of the program, sufficient to show safety instead of considering all the details present in the original program. Details of our algorithm are discussed in Chapter 3.

1.2 Safety of Probabilistic Transition Systems

Probabilistic systems are increasingly used for the formal modeling and analysis of a wide variety of systems ranging from randomized communication and security protocols to nanoscale computers and biological processes. There exist algorithms for model checking probabilistic systems against temporal logic specifications [18]. However, when the systems are comprised of multiple parallel components, model checking suffers from the *state-space explosion* problem [39], where the state space of a concurrent system grows exponentially in the number of its components.

The *assume-guarantee paradigm* for compositional reasoning [98] addresses this problem by separately verifying parts of the system using assumptions about the

environment, without verifying the whole system directly. For a system of two components, such reasoning is captured by the following simple assume-guarantee rule.

$$\frac{1 : \langle A \rangle\ L_1\ \langle P \rangle \qquad 2 : \langle true \rangle\ L_2\ \langle A \rangle}{\langle true \rangle\ L_1 \parallel L_2\ \langle P \rangle}\ (\textsc{Asym-Gen})$$

Here L_1 and L_2 are system components, P is a specification to be satisfied by the composite system $(L_1 \parallel L_2)$ and A is an assumption on L_1's environment, to be discharged on L_2. The challenge in using such an assume-guarantee rule is in coming up with a suitable intermediate assumption A automatically. Several other such rules have been proposed, some of them involving symmetric [99] or circular [9, 83, 99] reasoning. Despite its simplicity, rule Asym-Gen has been studied extensively in the context of non-probabilistic reasoning and is shown to be effective in inferring a suitable assumption automatically [31, 52, 99].

When the components L_1 and L_2 and the specification P are non-probabilistic and finite-state, there exist two different kinds of algorithms for inferring A. In the first kind, the algorithms adapt known automata learning techniques (e.g., [99]) and use positive and negative examples from both the premises. These are based on the *active learning* framework [14] where a *learner* tries to learn an unknown system/automaton based on the feedback given by a *teacher* in terms of positive and negative examples. In the assume-guarantee setting, the teacher is typically implemented by an algorithm that checks the premises of the rule and returns feedback correspondingly. In the second kind, the algorithms adapt the well-known CEGAR-loop [38] to the assume-

guarantee setting (e.g., [59]). These algorithms only utilize negative examples (i.e., counterexamples).

However, when the components L_1 and L_2 and the specification P are probabilistic, efficient algorithms are lacking for automating such an assume-guarantee framework. Existing techniques target *probabilistic reachability* properties and use algorithms based on automata learning [51, 52]. However, even when the problem is decidable monolithically, these algorithms are not guaranteed to terminate due to incompleteness of the inference rules [52] or due to the undecidability of checking the premises and of the learning algorithms used [51].

Given our primary interest in safety properties, we describe algorithms for checking *strong simulation* [102] between *Labeled Probabilistic Transition Systems* (LPTSes). Strong simulation is known to preserve the *weak safety* fragment of probabilistic CTL (PCTL), where the bound on the required probability of satisfaction of a CTL formula uses a non-strict inequality [29]. The corresponding instantiation of ASym-Gen is shown below, where $\preceq$ denotes the simulation conformance relation (see Chapter 4 for details) and the components L_1, L_2 and the specification P are all LPTSes.

$$\frac{1 : L_1 \parallel A \preceq P \qquad 2 : L_2 \preceq A}{L_1 \parallel L_2 \preceq P} \text{ (ASym)}$$

1.2.1 Counterexamples to Strong Simulation

One fundamental ingredient of an automatic framework for compositional verification is the use of *counterexamples* (from failed simulation checks) to iteratively refine inferred assumptions. However, to the best of our knowledge, the notion of a counterexample has not been previously formalized for strong simulation between LPTSes. We present a characterization of counterexamples to strong simulation as tree-shaped LPTSes and describe an algorithm to compute them.

1.2.2 Active Learning Based Approach

Our first set of algorithms is based on a framework for active learning to infer the unknown intermediate assumptions. We develop the first active learning framework for inferring an unknown LPTS (of minimal size) up to simulation equivalence (2-way simulation). We also discuss decidability results for inferring intermediate assumptions in ASym using the learning framework. Details are discussed in Chapter 5.

1.2.3 Abstraction-Refinement Based Approach

We also propose an *Assume-Guarantee Abstraction-Refinement* (AGAR) algorithm to automatically build the assumptions used in compositional reasoning. We first describe a CEGAR [38] based algorithm for strong simulation between LPTSes, which is then adapted to the compositional setting to obtain AGAR. Details are discussed in Chapter 6.

1.3 Experimental Results

We have implemented the model checking algorithms described in this thesis and analyzed the practical performance on realistic benchmarks. The implementations and benchmarks are available online.[5]

The algorithms for software model checking are implemented as part of the tool SPACER (which stands for *Software Proof-based Abstraction with CounterExample-based Refinement*) for verifying C programs. The back-end is based on the tool Z3 [45] which is used for SMT-solving and interpolation. It supports propositional logic, linear rational arithmetic, Presburger arithmetic, and bit-vectors (currently, via bit-blasting). The front-end is based on an existing tool called UFO [8] which converts C programs to the Horn-SMT format of Z3, corresponding to our logical program model.

We have a Java implementation of the algorithm AGAR for probabilistic systems. We also use the SMT solver Yices [46] for counterexample generation and checking strong simulation, and show experimentally that AGAR can achieve significantly better performance than monolithic verification.

[5] http://www.cs.cmu.edu/~akomurav/projects.

Chapter 2

SMT-Based Model Checking for Recursive Programs

2.1 Introduction

As mentioned in Chapter 1, several SMT-based algorithms exist for verifying safety of recursive programs. Notable examples are WHALE [7], HSF [63], GPDR[1] [74], Ultimate Automizer [68, 69] and Duality [92]. All of the algorithms are based on BMC for checking bounded safety for increasing values of the bound on the call-stack depth.[2] The use of BMC ensures that the algorithms are guaranteed to find a counterexample if the program fails to satisfy a safety property. However, with the exception of GPDR, the SMT problems created by these algorithms are monolithic, i.e., for the entire program, and the size of the problems can grow quite large. In par-

[1]GPDR stands for *Generalized Property Directed Reachability*.
[2]In this chapter, we assume that all loops have been turned into tail recursive procedures.

ticular, when the SMT problems correspond to a bounded call-stack depth, the size grows exponentially with the bound in the worst-case, due to the tree-like unrolling of the call-graph. Therefore, for the class of Boolean Programs, these algorithms are at least worst-case exponential in the number of states. However, as mentioned in Chapter 1, there exist model checking algorithms for safety of Boolean Programs that are polynomial (cubic) in the number of states [101]. The general observation behind these algorithms is that one can *summarize* the input-output behavior of a procedure, where a *summary* of a procedure is an input-output relation describing what is currently known about its behavior. Thus, if a summary has enough details, it can be used to analyze a procedure call without inlining [37, 103]. For a Boolean Program, the number of states is finite and hence, a summary can only be updated finitely many times. This observation led to a number of efficient algorithms that are polynomial in the number of states, e.g., the analysis framework by Reps, Horwitz, and Sagiv (RHS) [101], recursive state machines [10], BEBOP [19] and MOPED [49].

The SMT-based algorithms mentioned above also utilize summaries. If the monolithic SMT problems created are unsatisfiable, the current unwinding of the program is insufficient to find a counterexample. In this case, the algorithms use techniques based on Craig Interpolation [43] to obtain over-approximating summaries of procedures sufficient to show safety for the current unwinding. This is repeated with further unwindings of the program until a counterexample is found or the approximate summaries of the procedures are also invariant. However, as noted above, the size of the SMT problems created by these algorithms can grow exponentially.

On the other hand, the algorithm GPDR [74] follows the approach of IC3 [25]

by solving BMC incrementally without unrolling the call-graph. In GPDR, interpolation is used to obtain over-approximating summaries and partial models denoting undesirable reachable states are cached for future. For some configurations (e.g., explicit-state reasoning), GPDR is worst-case polynomial for Boolean Programs. However, it gets more challenging when the program operations and formulas are in a first-order language. In this case, GPDR might even fail to find a counterexample despite the presence of an SMT oracle, unlike the guarantee given by other BMC-based algorithms mentioned above (see Appendix 2.A for an example).

To address the aforementioned problems, we propose a new SMT-based algorithm RecMC for analyzing the program compositionally. That is, RecMC iteratively checks safety properties of individual procedures by inferring and utilizing approximating summaries of procedures. Our main insight is to maintain not only over-approximating summaries but also *under-approximating* summaries of the procedures. Syntactically, our approximations are assertions over the parameters of a procedure and auxiliary variables denoting the initial values of the parameters. Clarke showed that such assertions are sufficient to obtain a relatively complete Hoare proof system by making use of a *Rule of Adaptation* [37].

We use the terms *may-summary* and *must-summary*, respectively, to refer to such an over- and under-approximation. While may-summaries are used to block spurious counterexamples, must-summaries are used to analyze a procedure call without inlining the body of the callee. Thus, if the under-approximations given by the must-summaries can be reused at call-sites, they help avoid redundant explorations of the state-space. However, the must-summaries can be too strong to show falsification

and the may-summaries can be too weak to show satisfaction of a bounded safety property. In this case, our compositional algorithm creates and checks new bounded safety properties of the callee procedures and updates the approximations.

For Boolean Programs, as mentioned previously, the number of states is finite and hence, the approximations can only be updated finitely many times. As the approximations are reused at call-sites in a compositional manner, RecMC has a polynomial time complexity for Boolean Programs, by using an argument similar to that of RHS [101]. Moreover, assuming an SMT oracle for the first-order language of the assertions and the program operations, we show that RecMC terminates for bounded safety. To the best of our knowledge, ours is the first SMT-based algorithm with such guarantees.

Almost every step of RecMC introduces existential quantifiers in the assertions. RecMC tries to eliminate these quantified variables as, otherwise, they would accumulate exponentially in the value of the bound corresponding to the bounded safety problem. This is because, if no quantified variable is eliminated, the compositional algorithm essentially breaks down into an algorithm that unrolls the call-graph into a tree where, as we mentioned earlier, the size of the SMT problems created may grow exponentially in the bound on the call-stack. A naïve solution is to use quantifier elimination (QE), which results in an equivalent quantifier-free formula, but which is also expensive in practice. Instead, we develop an alternative approach that *under-approximates* QE, i.e., obtains a quantifier-free formula stronger than the original formula. However, obtaining arbitrary under-approximations can lead to divergence of the algorithm. We introduce the concept of *Model Based Projection* (MBP), for

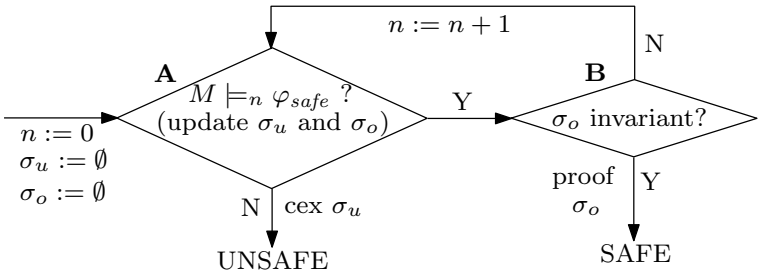


Figure 2.1: Flow of the algorithm RecMC to check if $M \models \varphi_{safe}$. σ_o and σ_u denote the may and must-summary maps.

covering $\exists \overline{x} \cdot \varphi(\overline{x}, \overline{y})$ by *finitely-many* quantifier-free under-approximations obtained using satisfying *models* of $\varphi(\overline{x}, \overline{y})$ (see Section 2.5). We developed efficient MBPs for Linear Rational Arithmetic (LRA) and Presburger Arithmetic (also known as Linear Integer Arithmetic (LIA)) based on the QE methods by Loos-Weispfenning [86] for LRA and Cooper [42] for LIA. We use MBP to under-approximate existential quantification in RecMC. In the best case, a partial under-approximation suffices and a complete quantifier elimination can be avoided.

In summary, we present: (a) an efficient, *compositional* SMT-based algorithm for model checking recursive programs, that uses under- and over-approximate summaries of procedure behavior (Section 2.4), (b) MBP functions for obtaining quantifier-free under-approximations of existential quantification for LRA and LIA (Section 2.5), (c) a new, complete algorithm for Boolean Programs, with complexity polynomial in the number of states, similar to the best known method [19] (see Section 2.4), and (d) an implementation and an empirical evaluation of the approach (Section 2.6).

2.2 Overview

In this section, we give an overview of RecMC and illustrate it on an example. Let $\mathcal{P}$ be a recursive program. We assume that there are no internal procedures and that procedures cannot be passed as parameters. Furthermore, for simplicity of presentation, assume no loops, no global variables and that arguments are passed by reference. Let $P(\overline{v}) \in \mathcal{P}$ be a procedure with parameters $\overline{v}$, and let $\overline{v}_0$ be fresh variables not appearing in P with $|\overline{v}_0| = |\overline{v}|$, denoting the initial values of $\overline{v}$. A safety property for P is an assertion $\varphi(\overline{v}_0, \overline{v})$ in a given assertion language with $\overline{v}_0$ and $\overline{v}$ as free variables. We say that P satisfies φ, denoted $P(\overline{v}) \models \varphi(\overline{v}_0, \overline{v})$, iff the Hoare-triple $\{\overline{v} = \overline{v}_0\}$ `call` $P(\overline{v})$ $\{\varphi(\overline{v}_0, \overline{v})\}$ is valid. Note that every Hoare-triple corresponds to a safety property in this sense, as shown by Clarke [37] using a *Rule of Adaptation*.

An execution of a procedure $P(\overline{v})$ is a sequence of valuations to the variables in scope, according to a given underlying semantics, beginning with an entry location of P and terminating with an exit location of P. We say that an execution satisfies an assertion $\varphi(\overline{v}_0, \overline{v})$ if assigning the initial and the final valuations of the execution to $\overline{v}_0$ and $\overline{v}$, respectively, makes φ true. For a natural number $n \geq 0$, we say that an execution *uses a call-stack bounded by* n if at no point during the execution there are more than n outstanding procedure calls that are not returned. We say that φ is a *bounded safety* property for P and n, denoted $P(\overline{v}) \models_n \varphi(\overline{v}_0, \overline{v})$, iff all executions of P using a call-stack bounded by n satisfy φ.

The key steps of RecMC are shown in Fig. 2.1. RecMC decides safety for the main procedure M of $\mathcal{P}$. RecMC maintains two *assertion maps* σ_u and σ_o. The

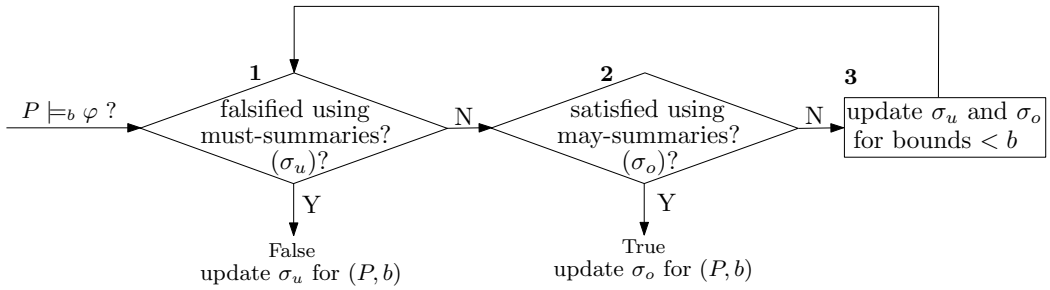


Figure 2.2: Flow of the algorithm BndSafety to check $P \models_b \varphi$.

must-summary map σ_u maps each procedure $P(\overline{v}) \in \mathcal{P}$ to a set of assertions over $\overline{v}_0 \cup \overline{v}$ that *under-approximate* its behavior. Similarly, the *may-summary* map σ_o maps a procedure P to a set of assertions that *over-approximate* its behavior. Given P, the maps are partitioned according to the bound on the call-stack. Therefore, if $\sigma_u(P, n)$, for some $n \geq 0$, contains an assertion $\delta(\overline{v}_0, \overline{v})$ with free variables $\overline{v}_0 \cup \overline{v}$, then δ under-approximates the combined behavior of all executions of P that use a call-stack of depth at most n. In other words, for every model m of δ, there is an execution of P that begins in $m(\overline{v}_0)$, the value of $\overline{v}_0$ under m, and ends in $m(\overline{v})$, the value of $\overline{v}$ under m, using a call-stack bounded by n. Similarly, if $\delta(\overline{v}_0, \overline{v}) \in \sigma_o(P, n)$, then δ over-approximates the behavior of all executions of P using a call-stack of depth at most n, i.e., $P(\overline{v}) \models_n \delta(\overline{v}_0, \overline{v})$.

RecMC alternates between two steps: (**A**) deciding bounded safety (that also updates σ_u and σ_o maps) and (**B**) checking whether the current proof of bounded safety also proves unbounded safety. It terminates when a counterexample or a proof is found.

Bounded safety, i.e., whether $P \models_b \varphi$, is decided using the algorithm BndSafety shown in Fig. 2.2. Step **1** checks whether φ is falsified using the current must-

```
                        T (t) {
M (m) {                     if (t>0) {
    T (m);                      t := t-2;          D (d) {
    D (m);                      T (t);                 d := d-1;
    D (m);                      t := t+1;          }
}                           }
                        }
```

Figure 2.3: A recursive program with 3 procedures.

summaries (σ_u) of the callees of P at bound $b-1$. If so, it infers a new must-summary for P at bound b witnessing the falsification of φ. Step **2** checks whether φ is satisfied using the current may-summaries (σ_o) of the callees at bound $b-1$. If so, it infers a new may-summary for P at bound b witnessing the satisfaction of φ. If the prior two steps fail, there is a potential counterexample π in P where the must-summaries of the callees are too strong to witness π but the may-summaries are too weak to block it. Step **3** checks the feasibility of such a path π by creating new bounded safety properties for the callees of P at bound $b-1$, recursively checking the new properties, and updating the assertion maps.

We conclude this section with an illustration of RecMC on the program in Fig. 2.3 (adapted from [37]). The program has 3 procedures: the main procedure `M`, and procedures `T` and `D`. The procedure `M` calls `T` and `D`. The procedure `T` modifies its argument `t` and calls itself recursively. The procedure `D` decrements its argument `d`. Suppose that we want to check if the (main procedure of the) program satisfies the safety property $\varphi \equiv m_0 \geq 2m+4$. The assertion maps σ_u and σ_o are initially empty.

In the first iteration of RecMC, the bound n on the call-stack is 0, i.e., the bounded safety problem is to check whether all executions that do not have any procedure calls are safe. Given that the only path in `M` has procedure calls, no

<table>
<tr><td colspan="3">$M(m) \models_1 m_0 \geq 2m + 4$?</td></tr>
<tr><td rowspan="2">iteration 1</td><td colspan="2">check new property: $D(d) \models_0 \bot$?</td></tr>
<tr><td>iteration 1</td><td>False; update $\sigma_u(D, 0)$ with $(d = d_0 - 1)$</td></tr>
<tr><td rowspan="2">iteration 2</td><td colspan="2">check new property: $T(t) \models_0 t_0 \geq 2t$?</td></tr>
<tr><td>iteration 1</td><td>True; update $\sigma_o(T, 0)$ with $(t_0 \geq 2t)$</td></tr>
<tr><td rowspan="2">iteration 3</td><td colspan="2">check new property: $D(d) \models_0 d \leq d_0 - 1$?</td></tr>
<tr><td>iteration 1</td><td>True; update $\sigma_o(D, 0)$ with $(d \leq d_0 - 1)$</td></tr>
<tr><td>iteration 4</td><td colspan="2">True; update $\sigma_o(M, 1)$ with $(m_0 \geq 2m + 4)$</td></tr>
</table>

Figure 2.4: A run of BndSafety for the program in Fig. 2.3 and the bounded safety property $M(m) \models_1 m_0 \geq 2m + 4$.

such executions exist and safety trivially holds for bound 0. Fig. 2.4 shows the four iterations of BndSafety for the next bound $n = 1$, i.e., for checking whether $M(m) \models_1 \varphi$ holds or not. In the first iteration of BndSafety, the current may and must-summaries of the callees are insufficient to satisfy or falsify the property, and there is a potential counterexample along the only path in `M`. Next, we create a new property for a callee, by performing a backward analysis along the potential counterexample path beginning with the negation of the safety property, and making use of the current summaries of the callees. In practice, one need not be restricted to a backward analysis; see Sections 2.4 and 2.6 for details. As shown in Fig. 2.4, assume that a new bounded safety property is created for `D` and $\sigma_u(\texttt{D}, 0)$, the must-summary map of `D` at bound 0, is updated with a new must-summary that witnesses the falsification of the property. In the second iteration of BndSafety, the current summaries are still insufficient and assume that a new property is created for `T` and $\sigma_o(\texttt{T}, 0)$ is updated with a new may-summary that witnesses the satisfaction of the property. To create the new property for `T`, we make use of the must-summary of `D` computed in the previous iteration for *both* the calls to `D` in `M`. This is where the compositionality of the algorithm helps avoid the potential re-computation of the

must-summary of D. Similarly, in the third iteration of BndSafety, let $\sigma_o(\mathtt{D}, 0)$ be updated with a new may-summary. At this point, the may-summaries for T and D at bound 0 are sufficient to establish bounded safety at $n = 1$ in the fourth iteration of BndSafety, resulting in an update of $\sigma_o(\mathtt{M}, 1)$.

Now, the may-summary map σ_o is:

$$\sigma_o(\mathtt{M}, 1) = \{m_0 \geq 2m + 4\}, \quad \sigma_o(\mathtt{T}, 0) = \{t_0 \geq 2t\}, \quad \sigma_o(\mathtt{D}, 0) = \{d \leq d_0 - 1\}$$

Ignoring the bounds, the may-summaries are invariant. For example, we can prove that the body of T satisfies $t_0 \geq 2t$, assuming that the calls do, i.e.,

$$\{t = t_0\}\ \mathtt{T}(t)\ \{t_0 \geq 2t\} \vdash \{t = t_0\}\ \mathit{Body}(\mathtt{T})\ \{t_0 \geq 2t\},$$

where *Body* denotes the body of a procedure. Thus, step **B** of RecMC succeeds and the algorithm terminates declaring the program SAFE.

In summary, RecMC checks safety of a recursive program in a compositional manner by inferring under- and over-approximations of the behavior of procedures. We use an SMT-solver for automating the steps of RecMC and BndSafety.

2.3 Preliminaries

Consider a first-order language with equality and let $\mathcal{S}$ be its signature, i.e., the set of non-logical function and predicate symbols (including equality). An $\mathcal{S}$-*structure* I consists of a domain of interpretation, denoted $|I|$, and assigns elements of $|I|$ to

variables, and functions and predicates on $|I|$ to the symbols of $\mathcal{S}$. Let φ be a formula in the first-order language. We assume the usual definition of satisfaction of φ by I, denoted $I \models \varphi$. I is called a *model* of φ iff $I \models \varphi$ and this can be extended to a set of formulas. A first-order $\mathcal{S}$-*theory* Th is a set of $\mathcal{S}$-sentences. I satisfies φ modulo Th, denoted $I \models_{Th} \varphi$, iff $I \models Th \cup \{\varphi\}$. φ is *valid* modulo Th, denoted $\models_{Th} \varphi$, iff every model of Th is also a model of φ.

Let I be an $\mathcal{S}$-structure and $\overline{w}$ be a list of fresh function/predicate symbols not in $\mathcal{S}$. A $(\mathcal{S} \cup \overline{w})$-structure J is called an *expansion* of I to $\overline{w}$ iff $|J| = |I|$ and J agrees with I on the assignments to all variables and the symbols of $\mathcal{S}$. We use the notation $I\{\overline{w} \mapsto \overline{u}\}$ to denote the expansion of I to $\overline{w}$ that assigns the function/predicate u_i to the symbol w_i. For an $\mathcal{S}$-sentence φ, we write $I(\varphi)$ to denote the truth value of φ under I. For a formula $\varphi(\overline{x})$ with a fixed ordering of the free variables $\overline{x}$, we overload the notation $I(\varphi)$ to mean $\{\overline{a} \in |I|^{|\overline{x}|} \mid I\{\overline{x} \mapsto \overline{a}\} \models \varphi\}$. For simplicity of presentation, we sometimes identify the truth value *true* with $|I|$ and *false* with $\emptyset$.

We assume that programs do not have internal procedures and that procedures cannot be passed as parameters. Furthermore, without loss of generality, we assume that programs do not have loops or global variables. In the following, we define programs using a logical representation, as opposed to giving a concrete syntax.

Definition 1 (Programs and Procedures)**.** *A* program $\mathcal{P}$ *is a finite list of procedures with a designated* main *procedure* M *where the program begins. A* procedure P *is a tuple* $\langle \overline{\iota}_P, \overline{o}_P, \Sigma_P, \overline{\ell}_P, \beta_P \rangle$*, where*

1. $\overline{\iota}_P$*,* $\overline{o}_P$*, and* $\overline{\ell}_P$ *are disjoint finite lists of variables denoting the input values of the parameters, the output values of the parameters, and the local variables,*

respectively,

2. Σ_P *is a fresh predicate symbol of arity* $|\overline{\iota}_P| + |\overline{o}_P|$,
3. β_P *is a quantifier-free sentence over the signature* $(\mathcal{S} \cup \{\Sigma_Q \mid Q \in \mathcal{P}\} \cup \overline{\iota}_P \cup \overline{o}_P \cup \overline{\ell}_P)$ *denoting the body of the procedure, where a predicate symbol* Σ_Q *appears only positively, i.e., under even number of negations.*

We use $\overline{v}_P$ *to denote* $\overline{\iota}_P \cup \overline{o}_P$.

Intuitively, for a procedure P, Σ_P is used to denote its semantics and β_P encodes its body using the predicate symbol Σ_Q for a call to the procedure Q. We require that a predicate symbol Σ_Q appears only positively in β_P to ensure a fixed-point characterization of the semantics as shown later on. For example, for the signature $\mathcal{S} = \langle 0, Succ, -, +, \leq, >, = \rangle$, the program in Fig. 2.3 is represented as $\langle M, T, D \rangle$ with the main procedure $M = \langle m_0, m, \Sigma_M, \langle \ell_0, \ell_1 \rangle, \beta_M \rangle$, $T = \langle t_0, t, \Sigma_T, \langle \ell_0, \ell_1 \rangle, \beta_T \rangle$, and $D = \langle d_0, d, \Sigma_D, \emptyset, \beta_D \rangle$, where

$$\begin{aligned} &\beta_M = \Sigma_T(m_0, \ell_0) \wedge \Sigma_D(\ell_0, \ell_1) \wedge \Sigma_D(\ell_1, m) \qquad \beta_D = (d = d_0 - 1) \\ &\beta_T = (t_0 \leq 0 \wedge t_0 = t) \;\; \vee \;\; (t_0 > 0 \wedge \ell_0 = t_0 - 2 \wedge \Sigma_T(\ell_0, \ell_1) \wedge t = \ell_1 + 1) \end{aligned} \tag{2.1}$$

Here, we abbreviate $Succ^i(0)$ by i and (m_0, t_0, d_0) and (m, t, d) denote the input and the output values of the parameters of the original program, respectively. For a procedure P, let $Paths(P)$ denote the set of all prime-implicants of β_P. Intuitively, each element of $Paths(P)$ encodes a path in the procedure.

Let $\mathcal{P} = \langle P_0, \ldots, P_n \rangle$ be a program and I be an $\mathcal{S}$-structure. Let $\overline{X}$ be a list of length n such that each X_i is either (i) a truth value if P_i has no parameters, i.e., $|\overline{v}_{P_i}| = 0$, or (ii) a subset of tuples from $|I|^{|\overline{v}_{P_i}|}$ if $|\overline{v}_{P_i}| \geq 1$. Let $J(I, \overline{X})$ denote

the expansion $I\{\Sigma_{P_0} \mapsto X_0\} \ldots \{\Sigma_{P_n} \mapsto X_n\}$. The *semantics* of a procedure P_i given I, denoted $[\![P_i]\!]_I$, characterizes all the terminating executions of P_i and is defined as follows. $\langle [\![P_0]\!]_I, \ldots, [\![P_n]\!]_I \rangle$ is the (point-wise) least $\overline{X}$ such that for all $Q \in \mathcal{P}$, $J(I, \overline{X}) \models \forall \overline{v}_Q \cup \overline{\ell}_Q \cdot (\beta_Q \implies \Sigma_Q(\overline{v}_Q))$. This has a well-known least fixed-point characterization [37].

For a natural number $b \geq 0$, denoting a bound on the call-stack, the *bounded semantics* of a procedure P_i given I, denoted $[\![P_i]\!]^b_I$, characterizes all the executions using a stack of depth bounded by b and is defined by induction on b:

$$[\![P_i]\!]^0_I = J(I, \langle \emptyset, \ldots, \emptyset \rangle)(\exists \overline{\ell}_{P_i} \cdot \beta_{P_i}),$$
$$[\![P_i]\!]^b_I = J(I, \langle [\![P_0]\!]^{b-1}_I, \ldots, [\![P_n]\!]^{b-1}_I \rangle)(\exists \overline{\ell}_{P_i} \cdot \beta_{P_i}),\ b > 0$$

Intuitively, $[\![P_i]\!]^0_I$ consists of all input-output values of the parameters of P_i reachable along paths that do not make any procedure calls, i.e., by interpreting every predicate symbol Σ_Q in the body β_{P_i} as $\emptyset$. Similarly, $[\![P_i]\!]^b_I$, for $b > 0$, consists of all input-output values of the parameters reachable along paths that use a stack of depth bounded by b.

An *environment* is a function that maps a predicate symbol Σ_P to a formula over $\overline{v}_P$. Given a formula τ and an environment E, we abuse the notation $[\![\cdot]\!]$ and write $[\![\tau]\!]_E$ for the formula obtained by instantiating every predicate symbol Σ_P by $E(\Sigma_P)$ in τ.

Let Th be an $\mathcal{S}$-theory. A *safety property* for a procedure $P \in \mathcal{P}$ is a formula over $\overline{v}_P$. We only consider safety properties that are quantifier-free or have one block

of existential quantifiers. P satisfies a safety property φ w.r.t Th, denoted $P \models_{Th} \varphi$, iff for all models I of Th, $[\![P]\!]_I \subseteq I(\varphi)$. A *safety property* ψ of the main procedure M of a program $\mathcal{P}$ is also called a safety property of the program itself. Given a safety property $\psi(\overline{v}_M)$, a *safety proof* for ψ is an environment Π that is both safe and invariant:

$$\models_{Th} [\![\forall \overline{x} \cdot \Sigma_M(\overline{x}) \implies \psi(\overline{x})]\!]_\Pi \text{ (safety)} \tag{2.2}$$

$$\forall P \in \mathcal{P} \cdot \models_{Th} [\![\forall \overline{v}_P \cup \overline{\ell}_P \cdot (\beta_P \implies \Sigma_P(\overline{v}_P))]\!]_\Pi \text{ (invariance)} \tag{2.3}$$

Given a safety property $\varphi(\overline{v}_P)$ and a natural number $b \geq 0$, denoting a bound on the call-stack, a procedure P satisfies *bounded safety* w.r.t Th, denoted $P \models_{b,Th} \varphi$, iff for all models I of Th, $[\![P]\!]^b_I \subseteq I(\varphi)$. In this case, we also call φ a *may-summary* for $\langle P, b\rangle$. We call φ a *must-summary* for $\langle P, b\rangle$ iff $I(\varphi) \subseteq [\![P]\!]^b_I$, for all models I of Th. Intuitively, *may-summaries* and *must-summaries* for $\langle P, b\rangle$, respectively, over- and under-approximate $[\![P]\!]^b_I$ for every model I of Th.

A *bounded assertion map* maps a procedure P and a natural number $b \geq 0$ to a set of formulas over $\overline{v}_P$. Given a bounded assertion map m and $b \geq 0$, we define two special environments U^b_m and O^b_m as follows.

$$U^b_m : \Sigma_P \mapsto \bigvee \{\delta \in m(P, b') \mid b' \leq b\} \qquad O^b_m : \Sigma_P \mapsto \bigwedge \{\delta \in m(P, b') \mid b' \geq b\}$$

We use U^b_m and O^b_m to under- and over-approximate the bounded semantics. For convenience, let U^{-1}_m and O^{-1}_m be environments that map every symbol to $\bot$.

```
RecMC(P, φ_safe)
1   n ← 0 ; σ_u ← ∅ ; σ_o ← ∅
2   while true do
3       res, σ_u, σ_o ← BndSafety(P, φ_safe, n, σ_u, σ_o)
4       if res is UNSAFE then
5           return UNSAFE, σ_u
6       else
7           ind, σ_o ← CheckInvariance(P, σ_o, n)
8           if ind then
9               return SAFE, σ_o
10          n ← n + 1

CheckInvariance(P, σ_o, n)
11  ind ← true
12  foreach P ∈ P do
13      foreach δ ∈ σ_o(P, n) do
14          if ⊨ [[β_P]]^n_o ⟹ δ then
15              σ_o ← σ_o ∪ (⟨P, n + 1⟩ ↦ δ)
16          else
17              ind ← false
18  return (ind, σ_o)
```

Figure 2.5: Pseudo-code of RecMC.

2.4 Model Checking Recursive Programs

In this section, we present our algorithm RecMC$(\mathcal{P}, \varphi_{safe})$ for determining whether a program $\mathcal{P}$ satisfies a safety property φ_{safe}. Let $\mathcal{S}$ be the signature of the first-order language under consideration and assume a fixed $\mathcal{S}$-theory Th. To avoid clutter, we drop the subscript Th from the notation $\models_{Th}$ and $\models_{b,Th}$. We also show the soundness of RecMC and discuss its complexity guarantees. An efficient instantiation of RecMC to Linear Arithmetic is presented in Section 2.5.

Top-level Loop. RecMC maintains two *bounded assertion maps* σ_u and σ_o for must and may-summaries, respectively. For brevity, for a first-order formula τ, we write $[\![\tau]\!]^b_u$ and $[\![\tau]\!]^b_o$ to denote $[\![\tau]\!]_{U^b_{\sigma_u}}$ and $[\![\tau]\!]_{O^b_{\sigma_o}}$, respectively, where the environments U^b_m and O^b_m, for a bounded assertion map m, are as defined in Section 2.3. Intuitively, $[\![\tau]\!]^b_u$ and $[\![\tau]\!]^b_o$, respectively, under- and over-approximate τ using σ_u and σ_o.

The pseudo-code of the main loop of RecMC (corresponding to the flow diagram in Fig. 2.1) is shown in Fig. 2.5. RecMC follows an *iterative deepening* strategy. In each iteration, BndSafety (described below) checks whether all executions of $\mathcal{P}$ satisfy φ_{safe} for a bound $n \geq 0$ on the call-stack, i.e., if $M \models_n \varphi_{safe}$. BndSafety also updates the maps σ_u and σ_o. Whenever BndSafety returns *UNSAFE*, the must-summaries in σ_u are sufficient to construct a counterexample to safety and the loop terminates. Whenever BndSafety returns *SAFE*, the may-summaries in σ_o are sufficient to prove the absence of a counterexample for the current bound n on the call-stack. In this case, if σ_o is also invariant (see (2.3)), as determined by CheckInvariance, $O^n_{\sigma_o}$ is a safety proof and the loop terminates. Otherwise, the bound on the call-stack is incremented and a new iteration of the loop begins. Note that, as a side-effect of CheckInvariance, some may-summaries are propagated to the bound $n+1$. This is similar to the *push generalization* phase in the IC3 algorithm [25].

Bounded Safety. We describe the routine BndSafety$(\mathcal{P}, \varphi_{safe}, n, \sigma_u^{Init}, \sigma_o^{Init})$ as an abstract transition system [94] defined by the inference rules shown in Fig. 2.6. Here, n is the current bound on the call-stack, and σ_u^{Init} and σ_o^{Init} are the maps of

$$\textsc{Init}\ \frac{}{\{\langle M, \neg\varphi_{safe}, n\rangle\} \parallel \sigma_u^{Init} \parallel \sigma_o^{Init}}$$

$$\textsc{May}\ \frac{\mathcal{Q} \parallel \sigma_u \parallel \sigma_o \qquad \langle P, \varphi, b\rangle \in \mathcal{Q} \qquad \models [\![\beta_P]\!]_o^{b-1} \implies \neg\varphi}{\mathcal{Q} \setminus \{\langle P, \eta, c\rangle \mid c \leq b, \models [\![\Sigma_P]\!]_o^c \wedge \psi \implies \neg\eta\} \parallel \sigma_u \parallel \sigma_o \cup \{\langle P, b\rangle \mapsto \psi\}}$$

$$\text{where } \psi = \textsc{Itp}([\![\beta_P]\!]_o^{b-1}, \neg\varphi)$$

$$\textsc{Must}\ \frac{\mathcal{Q} \parallel \sigma_u \parallel \sigma_o \qquad \langle P, \varphi, b\rangle \in \mathcal{Q} \qquad \pi \in \mathit{Paths}(P) \qquad \not\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi}{\mathcal{Q} \setminus \{\langle P, \eta, c\rangle \mid c \geq b, \not\models \psi \implies \neg\eta\} \parallel \sigma_u \cup \{\langle P, b\rangle \mapsto \psi\} \parallel \sigma_o}$$

$$\text{where } \psi = \exists \overline{\ell}_P \cdot [\![\pi]\!]_u^{b-1}$$

$$\textsc{Query}\ \frac{\begin{array}{c}\mathcal{Q} \parallel \sigma_u \parallel \sigma_o \qquad \langle P, \varphi, b\rangle \in \mathcal{Q} \qquad \models [\![\beta_P]\!]_u^{b-1} \implies \neg\varphi \qquad \pi \in \mathit{Paths}(P) \\ \pi = \pi_{pre} \wedge \Sigma_R(\overline{a}) \wedge \pi_{suf} \qquad \models [\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\Sigma_R(\overline{a})]\!]_u^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \implies \neg\varphi \\ \not\models [\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\Sigma_R(\overline{a})]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \implies \neg\varphi\end{array}}{\mathcal{Q} \cup \{\langle R, \psi, b-1\rangle\} \parallel \sigma_u \parallel \sigma_o}$$

$$\text{where } \begin{cases} \psi = \left(\exists \left(\overline{v}_P \cup \overline{\ell}_P\right) \setminus \overline{a} \cdot [\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \wedge \varphi\right)[\overline{a} \leftarrow \overline{v}_R] \\ \text{for all } \langle R, \eta, b-1\rangle \in \mathcal{Q}, \models \psi \implies \neg\eta \end{cases}$$

$$\textsc{Unsafe}\ \frac{\emptyset \parallel \sigma_u \parallel \sigma_o \qquad \not\models [\![\Sigma_M]\!]_u^n \implies \varphi_{safe}}{\mathit{UNSAFE}} \qquad \textsc{Safe}\ \frac{\emptyset \parallel \sigma_u \parallel \sigma_o \qquad \models [\![\Sigma_M]\!]_o^n \implies \varphi_{safe}}{\mathit{SAFE}}$$

Figure 2.6: Rules defining $\textsc{BndSafety}(\mathcal{P}, \varphi_{safe}, n, \sigma_u^{Init}, \sigma_o^{Init})$.

must and may-summaries input to the routine. A state of BndSafety is a triple $\mathcal{Q} \parallel \sigma_u \parallel \sigma_o$, where σ_u and σ_o are the current maps and $\mathcal{Q}$ is a set of triples $\langle P, \varphi, b\rangle$ for a procedure P, a formula φ over $\overline{v}_P$, and a number $b \geq 0$. A triple $\langle P, \varphi, b\rangle \in \mathcal{Q}$ is called a *bounded reachability query* and asks whether $P \not\models_b \neg\varphi$, i.e., whether there is an execution in P using a call-stack bounded by b where the values of $\overline{v}_P$ satisfy φ.

BndSafety starts with a single query $\langle M, \neg\varphi_{safe}, n\rangle$ and initializes the maps of must and may-summaries (rule Init). It checks whether $M \models_n \varphi_{safe}$ by generating new queries as necessary (rule Query) and answering existing queries using existing summaries (rules May and Must), the latter resulting in new summaries. When

there are no queries left to answer, i.e., $\mathcal{Q}$ is empty, BndSafety terminates with a result of either *UNSAFE* or *SAFE* (rules Unsafe and Safe). We explain the rules May, Must and Query below.

May infers a new may-summary when a query $\langle P, \varphi, b\rangle$ can be answered negatively. In this case, there is an over-approximation of the bounded semantics of P at bound b, obtained using the may-summaries of callees at bound $b-1$, that is unsatisfiable with φ. That is, $\models [\![\beta_P]\!]_o^{b-1} \implies \neg\varphi$. The inference of the new summary is by interpolation [43] (denoted by Itp in the side-condition of the rule). Thus, the new may-summary ψ is a formula over $\overline{v}_P$ such that $\models \big([\![\beta_P]\!]_o^{b-1} \implies \psi(\overline{v}_P)\big) \wedge (\psi(\overline{v}_P) \implies \neg\varphi)$. Note that ψ over-approximates the bounded semantics of P at b. Every query $\langle P, \eta, c\rangle \in \mathcal{Q}$ such that η is unsatisfiable with the updated environment $O^c_{\sigma_o}(\Sigma_P)$ is immediately answered and removed.

Must infers a new must-summary when a query $\langle P, \varphi, b\rangle$ can be answered positively. In this case, there is an under-approximation of the bounded semantics of P at b, obtained using the must-summaries of callees at bound $b-1$, that is satisfiable with φ. That is, $\not\models [\![\beta_P]\!]_u^{b-1} \implies \neg\varphi$. In particular, there exists a path π in $Paths(P)$ such that $\not\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi$. The new must-summary ψ is obtained by choosing such a path π non-deterministically and existentially quantifying all local variables from $[\![\pi]\!]_u^{b-1}$. Note that ψ under-approximates the bounded semantics of P at b. Every query $\langle P, \eta, c\rangle \in \mathcal{Q}$ such that η is satisfiable with the updated environment $U^c_{\sigma_u}(\Sigma_P)$ is immediately answered and removed.

Query creates a new query when an existing query $\langle P, \varphi, b\rangle$ cannot be answered using current summary maps σ_u and σ_o. In this case, the current over-approximation

of the bounded semantics of P at b is satisfiable with φ while its current under-approximation is unsatisfiable with φ. That is, $\not\models [\![\beta_P]\!]_o^{b-1} \implies \neg\varphi$ and $\models [\![\beta_P]\!]_u^{b-1} \implies \neg\varphi$. In particular, there exists a path π in $Paths(P)$ such that $\not\models [\![\pi]\!]_o^{b-1} \implies \neg\varphi$ and $\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi$. Intuitively, π is a potential counterexample path that needs to be checked for feasibility. Such a path π is chosen non-deterministically. π is guaranteed to have a conjunct $\Sigma_R(\overline{a})$, corresponding to a call to some procedure R, such that the under-approximation $[\![\Sigma_R(\overline{a})]\!]_u^{b-1}$ is too strong to witness an execution along π that satisfies φ but the over-approximation $[\![\Sigma_R(\overline{a})]\!]_o^{b-1}$ is too weak to block such an execution. That is, π can be partitioned into a prefix π_{pre}, a conjunct $\Sigma_R(\overline{a})$ corresponding to a call to R, and a suffix π_{suf} such that the following hold:

$$\models [\![\Sigma_R(\overline{a})]\!]_u^{b-1} \implies \left(([\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1}) \implies \neg\varphi\right) \tag{2.4}$$

$$\not\models [\![\Sigma_R(\overline{a})]\!]_o^{b-1} \implies \left(([\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1}) \implies \neg\varphi\right) \tag{2.5}$$

Note that the prefix π_{pre} and the suffix π_{suf} are over- and under-approximated, respectively. A new query $\langle R, \psi, b-1\rangle$ is created where ψ is obtained by existentially quantifying all variables from $[\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \wedge \varphi$ except the arguments $\overline{a}$ of the call, and renaming appropriately. If the new query is answered negatively (using May), all executions along π where the values of $\overline{v}_P \cup \overline{\ell}_P$ satisfy $[\![\pi_{suf}]\!]_u^{b-1}$ are spurious counterexamples. An additional side-condition requires that ψ "does not overlap" with η for any other query $\langle R, \eta, b-1\rangle$ in $\mathcal{Q}$. This is necessary for termination of BndSafety (Theorem 2). In practice, the side-condition is trivially satisfied by

	π_i	$[\![\pi_i]\!]^0_u$	$[\![\pi_i]\!]^0_o$
$i=1$	$\Sigma_T(m_0,\ell_0)$	$\bot$	$\top$
$i=2$	$\Sigma_D(\ell_0,\ell_1)$	$\ell_1 = \ell_0 - 1$	$\top$
$i=3$	$\Sigma_D(\ell_1,m)$	$m = \ell_1 - 1$	$\top$

Figure 2.7: Approximations of the only path π of the procedure M in Fig. 2.3.

always applying the rule to $\langle P, \varphi, b\rangle$ with the smallest b.

For example, consider the program in Fig. 2.3 represented by (2.1) and the query $\langle M, \varphi, 1\rangle$ where $\varphi \equiv m_0 < 2m + 4$. Let $\sigma_o = \emptyset$, $\sigma_u(D, 0) = \{d = d_0 - 1\}$ and $\sigma_u(T, 0) = \emptyset$. Let $\pi = (\Sigma_T(m_0, \ell_0) \wedge \Sigma_D(\ell_0, \ell_1) \wedge \Sigma_D(\ell_1, m))$ denote the only path in the procedure M. Fig. 2.7 shows $[\![\pi_i]\!]^0_u$ and $[\![\pi_i]\!]^0_o$ for each conjunct π_i of π. As the figure shows, $[\![\pi]\!]^0_o$ is satisfiable with φ, witnessed by the execution $e \equiv \langle m_0 = 3, \ell_0 = 3, \ell_1 = 2, m = 1\rangle$. Note that this execution also satisfies $[\![\pi_2 \wedge \pi_3]\!]^0_u$. But, $[\![\pi_1]\!]^0_u$ is too strong to witness it, where π_1 is the call $\Sigma_T(m_0, \ell_0)$. To create a new query for T, we first existentially quantify all variables other than the arguments m_0 and ℓ_0 from $\pi_2 \wedge \pi_3 \wedge \varphi$, obtaining $m_0 < 2\ell_0$. Renaming the arguments by the parameters of T results in the new query $\langle T, t_0 < 2t, 0\rangle$. Further iterations of BndSafety would answer this query negatively making the execution e spurious. Note that this would also make all other executions where the values to $\langle m_0, \ell_0, \ell_1, m\rangle$ satisfy $[\![\pi_2 \wedge \pi_3]\!]^0_u$ spurious.

Remark. Note that 2.4 above can be equivalently written as:

$$\models \left([\![\pi_{pre}]\!]^{b-1}_o \wedge [\![\pi_{suf}]\!]^{b-1}_u \wedge \varphi\right) \implies \neg[\![\Sigma_R(\overline{a})]\!]^{b-1}_u$$

Let $A = \left([\![\pi_{pre}]\!]^{b-1}_o \wedge [\![\pi_{suf}]\!]^{b-1}_u \wedge \varphi\right)$ and $B = \neg[\![\Sigma_R(\overline{a})]\!]^{b-1}_u$. So, the query created

by the rule QUERY is essentially the strongest interpolant for $A \implies B$. One can alternatively consider other interpolants as candidates for new queries. For example, the weakest interpolant B is another candidate for the new query. However, B is independent of A and is not property-driven. We leave these considerations for future exploration.

2.4.1 Soundness of BNDSAFETY and RECMC

Soundness of RECMC follows from that of BNDSAFETY, which can be shown by a case analysis on the inference rules.

Theorem 1. BNDSAFETY *and* RECMC *are sound.*

Proof. We only show the soundness of BNDSAFETY; the soundness of RECMC easily follows. In particular, for BNDSAFETY$(M, \varphi_{safe}, n, \emptyset, \emptyset)$ we show the following:

1. if the premises of UNSAFE hold, then $M \not\models_n \varphi_{safe}$, and
2. if the premises of SAFE hold, then $M \models_n \varphi_{safe}$.

It suffices to show that the environments $U^b_{\sigma_u}$ and $O^b_{\sigma_o}$, respectively, under- and over-approximate the bounded semantics of the procedures, for every $0 \leq b \leq n$. In particular, we show that the following is an invariant of BNDSAFETY: for every model I of the background theory Th, for every procedure $Q \in \mathcal{P}$ and $b \in [0, n]$,

$$I(U^b_{\sigma_u}(\Sigma_Q)) \subseteq [\![Q]\!]^b_I \subseteq I(O^b_{\sigma_o}(\Sigma_Q)). \tag{2.6}$$

Initially, σ_u and σ_o are empty and the invariant holds trivially. BNDSAFETY

updates σ_o and σ_u in the rules MAY and MUST, respectively. We show that these rules preserve (2.6). We only show the case of MAY. The case of MUST is similar.

Let $\langle P, \varphi, b\rangle \in \mathcal{Q}$ be such that MAY is applicable, i.e., $\models [\![\beta_P]\!]_o^{b-1} \implies \neg\varphi$. Let $\psi = \text{ITP}([\![\beta_P]\!]_o^{b-1}, \neg\varphi)$. Note that φ, and hence ψ, does not depend on the local variables $\overline{\ell}_P$. Hence, we know that

$$\models \left(\exists \overline{\ell}_P \cdot [\![\beta_P]\!]_o^{b-1}\right) \implies \psi. \tag{2.7}$$

The case of $b = 0$ is easy and we will skip it. Let I be an arbitrary model of Th. Assume that (2.6) holds at $b-1$ before applying the rule. In particular, assume that for all $Q \in \mathcal{P}$, $[\![Q]\!]_I^{b-1} \subseteq I(O_{\sigma_o}^{b-1}(\Sigma_Q))$.

We will first show that the new may-summary ψ over-approximates $[\![P]\!]_I^b$. Let $J(I, \overline{X})$ be an expansion of I as defined in Section 2.3.

$$\begin{aligned}
[\![P]\!]_I^b &= J(I, \langle [\![P_0]\!]_I^{b-1}, \ldots, [\![P_n]\!]_I^{b-1}\rangle)(\exists \overline{\ell}_{P_i} \cdot \beta_{P_i}) \\
&\subseteq J(I, \langle I(O_{\sigma_o}^{b-1}(\Sigma_{P_0})), \ldots, I(O_{\sigma_o}^{b-1}(\Sigma_{P_n}))\rangle)(\exists \overline{\ell}_{P_i} \cdot \beta_{P_i}) && \text{(hypothesis)} \\
&= I([\![\exists \overline{\ell}_P \cdot \beta_P]\!]_{O_{\sigma_o}^{b-1}}) && (O_{\sigma_o}^{b-1} \text{ is FO-definable}) \\
&= I(\exists \overline{\ell}_P \cdot [\![\beta_P]\!]_{O_{\sigma_o}^{b-1}}) && \text{(logic)} \\
&= I(\exists \overline{\ell}_P \cdot [\![\beta_P]\!]_o^{b-1}) && \text{(notation)} \\
&\subseteq I(\psi) && \text{(from (2.7))}
\end{aligned}$$

Next, we show that the invariant continues to hold. The map of may-summaries is

updated to $\sigma'_o = \sigma_o \cup \{\langle P, b\rangle \mapsto \psi\}$. Now, σ'_o differs from σ_o only for the procedure P and for bounds in $[0, b]$. Let $b' \in [0, b]$ be arbitrary. Since (2.6) was true before applying MAY, we know that $[\![P]\!]^{b'}_I \subseteq I(O^{b'}_{\sigma_o}(\Sigma_P))$. As $[\![P]\!]^{b'}_I \subseteq [\![P]\!]^{b}_I \subseteq I(\psi)$, it follows that $[\![P]\!]^{b'}_I \subseteq I(O^{b'}_{\sigma_o}(\Sigma_P)) \cap I(\psi) \subseteq I(O^{b'}_{\sigma_o}(\Sigma_P) \wedge \psi) = I(O^{b'}_{\sigma'_o}(\Sigma_P))$. □

2.4.2 Termination and Complexity of BNDSAFETY

We will now show that BNDSAFETY is complete relative to an oracle for satisfiability of existentially quantified formulas (i.e., formulas that are quantifier-free or have one block of existential quantifiers) modulo *Th*. Throughout the following, we assume that such an oracle exists. Intuitively, a must-summary inferred by BNDSAFETY corresponds to a path in a procedure and given a bound on the call-stack, the number of such formulas is finite. This bounds the number of may/must-summaries inferred by BNDSAFETY, guaranteeing termination.

The following lemma shows that when a query is removed from $\mathcal{Q}$, it is actually answered. The proof is immediate from the definitions of $O^b_{\sigma_o}$ and $U^b_{\sigma_u}$ given in Section 2.3.

Lemma 1 (Answered Queries)**.** *Whenever* BNDSAFETY *removes a query from* $\mathcal{Q}$*, it is answered using the known must and may-summaries. In particular, for every query* $\langle P, \eta, b\rangle \in \mathcal{Q}$ *removed from* $\mathcal{Q}$ *by* BNDSAFETY*,*

1. *if the query is removed by* MAY*, then* $\models [\![\Sigma_P]\!]^b_o \implies \neg\eta$*, and*
2. *if the query is removed by* MUST*, then* $\not\models [\![\Sigma_P]\!]^b_u \implies \neg\eta$*.*

Next, we show that current summaries are insufficient to answer existing queries in $\mathcal{Q}$.

Lemma 2 (Pending Queries)**.** *$\mathcal{Q}$ only has the queries which cannot be immediately answered by σ_u or σ_o, i.e., as long as $\langle P, \eta, \ell\rangle$ is in $\mathcal{Q}$, the following are invariants of* BndSafety*.*

1. $\not\models [\![\Sigma_P]\!]_o^\ell \implies \neg\eta$*, and*
2. $\models [\![\Sigma_P]\!]_u^\ell \implies \neg\eta$*.*

Proof. We first show that the invariants hold when a query is newly created by Query. Let P, η and ℓ be, respectively, R, $\psi[\overline{a} \leftarrow \overline{v}_R]$ and $b-1$, as in the conclusion of the rule. The last-but-one premise of Query is

$$\models [\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\Sigma_R(\overline{a})]\!]_u^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \implies \neg\varphi$$

which implies that

$$\models [\![\Sigma_R(\overline{a})]\!]_u^{b-1} \implies \neg\left([\![\pi_{pre}]\!]_o^{b-1} \wedge [\![\pi_{suf}]\!]_u^{b-1} \wedge \varphi\right).$$

The variables not in common, viz., $(\overline{v}_P \cup \overline{\ell}_P) \setminus \overline{a}$, can be universally quantified from the right hand side resulting in $\models [\![\Sigma_R]\!]_u^{b-1} \implies \neg\eta$. Similarly, $\not\models [\![\Sigma_R]\!]_o^{b-1} \implies \neg\eta$ follows from the last premise of the rule. Next, we show that May and Must preserve the invariants.

Let May answer a query $\langle P, \varphi, \ell\rangle$ with a new may-summary ψ and let the updated map of may-summaries be $\sigma'_o = \sigma_o \cup \{\langle P, \ell\rangle \mapsto \psi\}$. Now, consider $\langle P, \eta, \ell'\rangle \in \mathcal{Q}$ after the application of the rule. If $\ell' > \ell$, $O^{\ell'}_{\sigma'_o} = O^{\ell'}_{\sigma_o}$ and the invariant continues to hold. So, assume $\ell' \leq \ell$. From the conclusion of May, we have $\not\models [\![\Sigma_P]\!]_o^{\ell'} \wedge \psi \implies \neg\eta$.

Now, $O^{\ell'}_{\sigma'_o}(\Sigma_P) = O^{\ell'}_{\sigma_o}(\Sigma_P) \wedge \psi$. So, the invariant continues to hold.

Similarly, let MUST answer a query $\langle P, \varphi, \ell \rangle$ with a new must-summary ψ and let the updated map of must-summaries be $\sigma'_u = \sigma_u \cup \{\psi \mapsto \langle P, \ell \rangle\}$. Now, consider $\langle P, \eta, \ell' \rangle \in \mathcal{Q}$ after the application of the rule. If $\ell' < \ell$, $U^{\ell'}_{\sigma'_u} = U^{\ell'}_{\sigma_u}$ and the invariant continues to hold. So, assume $\ell' \geq \ell$. From the conclusion of MUST, we have $\models \psi \implies \neg\eta$. Assuming the invariant holds before the rule application, we also have $\models [\![\Sigma_P]\!]^{\ell'}_u \implies \neg\eta$. Therefore, we have $\models [\![\Sigma_P]\!]^{\ell'}_u \vee \psi \implies \neg\eta$. Now, $U^{\ell'}_{\sigma'_u}(\Sigma_P) = U^{\ell'}_{\sigma_u}(\Sigma_P) \vee \psi$. So, the invariant continues to hold. □

The next few lemmas show that the rules of the algorithm cannot be applied indefinitely, leading to a termination argument. Let N be the number of procedures in the program $\mathcal{P}$, p be the maximum number of paths in a procedure, and c be the maximum number of procedure calls along any path in $\mathcal{P}$.

Lemma 3 (Finitely-many Must Summaries)**.** *Given a predicate symbol Σ_P and a bound b, the environment $U^b_{\sigma_u}$ is updated only $O(N^b \cdot p^{b+1})$-many times.*

Proof. The environment $U^b_{\sigma_u}$ can be updated for Σ_P and b whenever a must-summary is inferred for P at a bound $b' \leq b$. Now, a must-summary is obtained per path (after eliminating the local variables) of a procedure, using the currently known must-summaries about the callees. Moreover, Lemmas 1 and 2 imply that no must-summary is inferred twice. This is because whenever a query is answered using MUST, the query could not have been answered using already existing must-summaries and a new must-summary is inferred.

This gives the following recurrence $\mathit{Must}(b)$ for the number of updates to $U^b_{\sigma_u}$ for

a given Σ_P:

$$Must(b) = \begin{cases} p, & b = 0 \\ (p \cdot N + 1) \cdot Must(b-1), & b > 0. \end{cases}$$

In words, for $b = 0$, the number of updates is given by the number of must-summaries that can be inferred, which is bounded by the number of paths p in the procedure P. For $b > 0$, the environment $U^b_{\sigma_u}$ is updated when a must-summary is learnt for the procedure at a bound smaller than or equal to b. For the former, the number of updates is simply $Must(b-1)$. For the latter, a new must-summary is inferred at bound b along a path whenever $U^{b-1}_{\sigma_u}$ changes for a callee. For N procedures and p paths, this is given by $(p \cdot N \cdot Must(b-1))$.

This gives us $Must(b) = O(N^b \cdot p^{b+1})$. □

Lemma 4 (Finitely-many Queries)**.** *For $\langle P, \varphi, b\rangle \in \mathcal{Q}$,* QUERY *is applicable only $O(c \cdot N^b \cdot p^{b+1})$-many times.*

Proof. First, assume that the environments $U^{b-1}_{\sigma_u}$ and $O^{b-1}_{\sigma_o}$ are fixed. The number of possible queries that can be created for a given path of P is bounded by the number of ways the path can be divided into a prefix, a procedure call, and a suffix. This is bounded by c, the maximum number of calls along the path. For p paths, this is bounded by $c \cdot p$.

Consider a path π and its division, and let a query be created for a callee R along π. Now, while the query is still in $\mathcal{Q}$, updates to the environments $O^{b-1}_{\sigma_o}$ and $U^{b-1}_{\sigma_u}$ do not result in a new query for R for the same division along π. This is because, the new query would overlap with the existing one and this is disallowed by the second

side-condition of QUERY.

Suppose that the new query is answered by MAY. With the updated map of may-summary, the last premise of QUERY can be shown to fail for the current division of π. If $O_{\sigma_o}^{b-1}$ is updated, the last premise continues to fail. So, a new query can be created for the same prefix and suffix along π only if $U_{\sigma_u}^{b-1}$ is updated for some callee along π. The other possibility is that the query is answered by MUST which updates $U_{\sigma_u}^{b-1}$ as well.

Thus, for a given path, and a given division of it into prefix and suffix, the number of queries that can be created is bounded by the number of updates to $U_{\sigma_u}^{b-1}$ which is $(N \cdot \mathit{Must}(b-1))$. Here, *Must* is as in Lemma 3. So, the number of times QUERY is applicable for a given query $\langle P, \varphi, b\rangle$ is $O(p{\cdot}c{\cdot}N{\cdot}\mathit{Must}(b{-}1))$. As $\mathit{Must}(b) = N^b{\cdot}p^{b+1}$, we obtain the bound $O(c \cdot N^b \cdot p^{b+1})$. □

Lemma 5 (Progress)**.** *As long as $\mathcal{Q}$ is non-empty, either* MAY*,* MUST *or* QUERY *is always applicable.*

Proof. First, we show that for every query in $\mathcal{Q}$, either of the three rules is applicable, without the second side-condition in QUERY. Let $\langle P, \varphi, b\rangle \in \mathcal{Q}$. If $\models [\![\beta_P]\!]_o^{b-1} \implies \neg\varphi$, then MAY is applicable. Otherwise, there exists a path $\pi \in \mathit{Paths}(P)$ such that $[\![\pi]\!]_o^{b-1}$ is satisfiable with φ, i.e., $\not\models [\![\pi]\!]_o^{b-1} \implies \neg\varphi$. Now, if $[\![\pi]\!]_u^{b-1}$ is also satisfiable with φ, i.e., $\not\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi$, MUST is applicable. Otherwise, $\models [\![\pi]\!]_u^{b-1} \implies \neg\varphi$. Note that this can only happen if $b > 0$, as otherwise, there will not be any procedure calls along π and $[\![\pi]\!]_o^{b-1}$ and $[\![\pi]\!]_u^{b-1}$ would be equivalent.

Let $\pi = \pi_0 \wedge \pi_1 \wedge \ldots \pi_l$ for some finite l. Then, $[\![\pi]\!]_o^{b-1}$ is obtained by taking the

conjunction of the formulas

$$\langle [\![\pi_0]\!]_o^{b-1}, [\![\pi_1]\!]_o^{b-1}, \dots \rangle.$$

Similarly, $[\![\pi]\!]_u^{b-1}$ is obtained by taking the conjunction of the formulas

$$\langle [\![\pi_0]\!]_u^{b-1}, [\![\pi_1]\!]_u^{b-1}, \dots \rangle.$$

From Theorem 1, we can think of obtaining the latter sequence of formulas by conjoining $[\![\pi_i]\!]_u^{b-1}$ to $[\![\pi_i]\!]_o^{b-1}$ for every i. When this is done backwards for decreasing values of i, an intermediate sequence looks like

$$\langle [\![\pi_0]\!]_o^{b-1}, \dots, [\![\pi_{j-1}]\!]_o^{b-1}, [\![\pi_j]\!]_u^{b-1} \dots \rangle.$$

As $[\![\pi]\!]_u^{b-1}$ is unsatisfiable with φ, there exists a maximal j such that the conjunction of constraints in such an intermediate sequence are unsatisfiable with φ. Moreover, π_j must be a literal of the form $\Sigma_R(\overline{a})$ as otherwise, $[\![\pi_j]\!]_o^{b-1} = [\![\pi_j]\!]_u^{b-1}$ violating the maximality condition on j. Thus, all premises of QUERY hold and the rule is applicable.

Now, the second side-condition in QUERY can be trivially satisfied by always choosing a query in $\mathcal{Q}$ with the smallest bound for the next rule to apply. This is because, if $\langle R, \eta, b-1 \rangle$ is the newly created query, there is no other query in $\mathcal{Q}$ for R and $b-1$. □

Lemmas 4 and 5 imply that every query in $\mathcal{Q}$ is eventually answered by MAY or

MUST, as shown below.

Lemma 6 (Eventual Answer)**.** *Every* $\langle P, \varphi, b\rangle \in \mathcal{Q}$ *is eventually answered by* MAY *or* MUST*, in* $O(b \cdot c^b \cdot (Np)^{O(b^2)})$ *applications of the rules.*

Proof. Firstly, to answer any given query in $\mathcal{Q}$, Lemma 4 guarantees that the algorithm can only create finitely many queries. Lemma 5 guarantees that some rule is always applicable, as long as $\mathcal{Q}$ is non-empty. Thus, when QUERY cannot be applied for any query in $\mathcal{Q}$, either MAY or MUST must be applicable for some query. Thus, eventually, all queries are answered.

The total number of rule applications to answer $\langle P, \varphi, b\rangle$ is then linear in the cumulative number of applications of QUERY, which has the following recurrence:

$$T(b) = \begin{cases} Q(0), & b = 0 \\ Q(b)(1 + T(b-1)), & b > 0. \end{cases}$$

where $Q(b)$ denotes the number of applications of QUERY for a fixed query in $\mathcal{Q}$ at bound b. From Lemma 4, $Q(b) = O(c \cdot N^b \cdot p^{b+1})$. This gives us $T(b) = O(b \cdot c^b \cdot (Np)^{O(b^2)})$. □

The main termination theorem is an immediate consequence of the above lemma:

Theorem 2. *Given an oracle for satisfiability of existentially quantified formulas modulo Th,* BNDSAFETY$(\mathcal{P}, \varphi, n, \emptyset, \emptyset)$ *decides bounded safety in finitely many iterations and terminates.*

As an immediate corollary, RECMC is guaranteed to find a counterexample if one exists.

Corollary 1. RecMC$(\mathcal{P}, \varphi)$ *is guaranteed to return UNSAFE with a counterexample if* $\mathcal{P} \not\models \varphi$.

In contrast, the closest related algorithm GPDR [74], mentioned briefly in Section 2.1, does not have such guarantees. Finally, for Boolean Programs RecMC is a complete decision procedure. Unlike the general case, the number of reachable states of a Boolean Program, and hence the number of summaries, is finite. Boolean programs are obtained when the signature $\mathcal{S}$ is assumed to be empty, i.e., there are no non-logical function or predicate symbols. Let N denote the number of procedures of a program $\mathcal{P}$ and $k = \max\{|\overline{v}_P| \mid P(\overline{v}_P) \in \mathcal{P}\}$.

Theorem 3. *Let* $\mathcal{P}$ *be a Boolean Program. Then* RecMC$(\mathcal{P}, \varphi)$ *terminates in* $O(N^2 \cdot 2^{2k})$*-many applications of the rules in Fig. 2.6.*

Proof. First, assume a bound n on the call-stack. The number of queries that can be created for a procedure at any given bound is $O(2^k)$, the number of possible valuations of the parameters (note that Query disallows overlapping queries to be present simultaneously in $\mathcal{Q}$). For N procedures and n possible values of the bound, the complexity of BndSafety$(\mathcal{P}, \varphi, n, \emptyset, \emptyset)$, for a Boolean Program, is $O(N \cdot 2^k \cdot n)$.

Now, the total number of may-summaries that can be inferred for a procedure is also bounded by $O(2^k)$. As $O^b_{\sigma_o}$ is monotonic in b, the number of iterations of RecMC is bounded by $O(N \cdot 2^k)$, the cumulative number of states of all procedures. Thus, we obtain the bound $O(N^2 \cdot 2^{2k})$ on the number of applications of the rules in Fig. 2.6. □

Note that the number of states of a Boolean Program is $O(N \cdot 2^k)$, so the above

bound is polynomial in the number of states. Moreover, assuming that we always eliminate Boolean existential quantifiers using quantifier elimination, the total complexity of RecMC is also polynomial in the number of states. In contrast, other SMT-based algorithms, such as Whale [7], are worst-case exponential in the number of states of a Boolean Program. Also, note that the complexity is quadratic in the number of procedures as opposed to the known upper-bound which has a linear dependency [19]. This is a manifestation of the iterative deepening strategy of RecMC and in particular, the may-summaries computed by the algorithm, which is necessary for handling programs over first-order theories. In contrast, the known optimal algorithms for Boolean programs do not compute may-summaries.

In summary, RecMC checks safety of a recursive program by inferring the necessary under- and over-approximations of procedure semantics and using them to analyze procedures individually.

2.5 Model Based Projection

The algorithm RecMC described in the previous section works for an arbitrary first-order signature $\mathcal{S}$ and a $\mathcal{S}$-theory Th as long as there is an oracle for satisfiability (of existentially quantified formulas) modulo Th. One can also use RecMC as-is for many-sorted signatures and a corresponding combination of theories, as long as there is an SMT oracle for existentially quantified formulas modulo the theory combination. In this section, we restrict ourselves to the two combinations of Propositional Logic with the theories of Linear Rational Arithmetic (LRA) and Linear Integer Arithmetic

(LIA) (also well known as Presburger Arithmetic), and let the corresponding sorts be `Bool`, `Rat`, and `Int`, respectively.

Even though RecMC can be used as-is for LRA and LIA, recall that BndSafety introduces quantifiers in the formulas maintained by the algorithm. This is because the may and must-summaries are formulas over the parameters of a procedure and auxiliary variables denoting their initial values, and when creating a new summary, all other variables will be quantified away. Same is the case with creating new bounded safety properties. If these quantifiers are not eliminated, every use of a summary at a call-site will introduce a different copy of the quantified variables which, in the worst-case, can end up accumulating exponentially in the bounded safety properties created by the algorithm. In essence, the compositional algorithm will break down into a non-compositional one, similar to unrolling the call-graph into a tree where, as we mentioned earlier, the size of the SMT problems created can grow exponentially in the bound on the call-stack. On the other hand, it is expensive to use quantifier elimination (QE) to obtain an equivalent quantifier-free formula. Instead, we propose an alternative approach that *approximates* QE with quantifier-free formulas *lazily* and efficiently.

In particular, we (a) introduce a model-based under-approximation of QE for existentially quantified formulas, called *Model Based Projection* (MBP), (b) give efficient (linear in the size of formulas involved) MBP procedures for Propositional Logic, LRA, and LIA, and (c) present a modified version of BndSafety that uses MBP to under-approximate the existential quantification of variables out of scope, and show that it remains sound and terminating. Our MBP procedures for LRA and

LIA are based on the QE algorithms by Loos and Weispfenning [86] and Cooper [42], respectively.

Definition 2 (Model Based Projection)**.** *Let $\eta(\overline{y}) = \exists \overline{x} \cdot \eta_m(\overline{x}, \overline{y})$ be an existentially quantified formula where η_m is quantifier free. A function $Proj_\eta$ from models of η_m to quantifier-free formulas over $\overline{y}$ is a* Model Based Projection *(for η) iff*

1. *$Proj_\eta$ has a finite image,*
2. *$\eta \equiv \bigvee_{M \models \eta_m} Proj_\eta(M)$, and*
3. *for every model M of η_m, $M \models Proj_\eta(M)$.*

In other words, $Proj_\eta$ covers the space of all models of $\eta_m(\overline{x}, \overline{y})$ by a finite set of quantifier-free formulas over $\overline{y}$. Note that there is a trivial MBP that maps every model of η_m to a quantifier-free formula equivalent to η. However, when QE is expensive, it is not the most efficient MBP and our objective is to obtain an MBP that maps models to quantifier-free *under-approximations* of η. In the following, we describe MBP procedures whose computation is linear in time and space given a model.

2.5.1 MBP for Propositional Logic

Let $\eta(\overline{y}) = \exists \overline{x} \cdot \eta_m(\overline{x}, \overline{y})$ be an existentially quantified formula where the quantified variables in $\overline{x}$ are all of sort `Bool`. Without loss of generality, assume that $\overline{x}$ is singleton. Our MBP procedure is based on the following equivalence:

$$\exists x \cdot \eta_m(x, \overline{y}) \;\equiv\; \eta_m[\bot] \vee \eta_m[\top] \tag{2.8}$$

where $[\cdot]$ denotes a substitution for x.

We now define an MBP $BoolProj_\eta$ for Propositional Logic as a map from models of η_m to one of the disjuncts above depending on the assignment to x in the given model M:

$$BoolProj_\eta(M) = \begin{cases} \eta_m[\bot], & M \models x = \bot \\ \eta_m[\top], & M \models x = \top \end{cases}$$

This procedure is also used in the GPDR model checking algorithm [74] implemented in the tool Z3 [45] and a similar approach is used in SAT-based iterative quantifier elimination in hardware verification [55]. The following is now immediate.

Theorem 4. *$BoolProj_\eta$ is a Model Based Projection.*

2.5.2 MBP for Linear Rational Arithmetic (LRA)

We begin with a brief overview of Loos-Weispfenning (LW) method [86] for quantifier elimination in LRA. We borrow our presentation from Nipkow [95] to which we refer the reader for more details. Let $\eta(\overline{y}) = \exists \overline{x} \cdot \eta_m(\overline{x}, \overline{y})$ as above, where the variables in $\overline{x}$ are of sort `Rat`. Let Th be LRA, or its combination with Propositional Logic. Without loss of generality, assume that $\overline{x}$ is singleton, η_m is in Negation Normal Form, and x only appears in the literals of the form $\ell < x$, $x < u$, and $x = e$, where ℓ, u, and e are x-free. Let $lits(\eta)$ denote the literals of η. The LW-method states that

$$\exists x \cdot \eta_m(x, \overline{y}) \equiv \left(\bigvee_{(x=e) \in lits(\eta)} \eta_m[e] \vee \bigvee_{(\ell < x) \in lits(\eta)} \eta_m[\ell + \epsilon] \vee \eta_m[-\infty] \right) \quad (2.9)$$

where $\eta_m[\cdot]$ denotes a *virtual substitution* for the literals containing x. Intuitively, $\eta_m[e]$ covers the case when a literal $(x = e)$ is true, $\eta_m[\ell + \epsilon]$ covers the case where ℓ is the largest lower bound satisfied by x, and $\eta_m[-\infty]$ covers the remaining cases. We omit the details of the substitution and instead illustrate it on an example. Let η_m be $(x = e \wedge \phi_1) \vee (\ell < x \wedge x < u) \vee (x < u \wedge \phi_2)$, where $\ell, e, u, \phi_1, \phi_2$ are x-free. Then,

$$\begin{aligned}
\exists x \cdot \eta_m &\equiv \eta_m[e] \vee \eta_m[\ell + \epsilon] \vee \eta_m[-\infty] \\
&\equiv \big(\phi_1 \vee (\ell < e \wedge e < u) \vee (e < u \wedge \phi_2)\big) \vee \big(\ell < u \vee (\ell < u \wedge \phi_2)\big) \vee \phi_2 \\
&\equiv \phi_1 \vee (\ell < u) \vee \phi_2
\end{aligned}$$

We now define an MBP $LRAProj_\eta$ for LRA as a map from models of η_m to disjuncts in (2.9). Given $M \models \eta_m$, $LRAProj_\eta$ picks a disjunct that covers M based on values of the literals of the form $x = e$ and $\ell < x$ in M. Ties are broken by a syntactic ordering on terms (e.g., when $M \models \ell' = \ell$ for two literals $\ell < x$ and $\ell' < x$).

$$LRAProj_\eta(M) = \begin{cases} \eta_m[e], & \text{if } (x = e) \in \mathit{lits}(\eta) \wedge M \models x = e \\ \eta_m[\ell + \epsilon], & \text{else if } (\ell < x) \in \mathit{lits}(\eta) \wedge M \models \ell < x \wedge \\ & \quad \forall (\ell' < x) \in \mathit{lits}(\eta) \cdot M \models ((\ell' < x) \implies (\ell' \leq \ell)) \\ \eta_m[-\infty], & \text{otherwise} \end{cases}$$

Theorem 5. *$LRAProj_\eta$ is a Model Based Projection.*

Proof. By definition, $LRAProj_\eta$ has a finite image, as there are only finitely many dis-

juncts in (2.9). Thus, it suffices to show that for every $M \models \eta_m$, $M \models LRAProj_\eta(M)$.

Each disjunct in the LW decomposition (2.9) is obtained by a *virtual substitution* of the literals in η_m containing x. As mentioned in the beginning of the section, we assume that η_m is in NNF with the only literals containing x of the form $(x = e)$, $(\ell < x)$ or $(x < u)$ for x-free terms e, ℓ and u. Let Sub_t denote the virtual substitution map of literals when t is either e, $\ell + \epsilon$ or $-\infty$. The LW method [86] defines:

$$Sub_e(x = e) = \top, Sub_e(\ell < x) = (\ell < e), Sub_e(x < u) = (e < u) \tag{2.10}$$

$$Sub_{\ell+\epsilon}(x = e) = \bot, Sub_{\ell+\epsilon}(\ell' < x) = (\ell' \leq \ell), Sub_{\ell+\epsilon}(x < u) = (\ell < u) \tag{2.11}$$

$$Sub_{-\infty}(x = e) = \bot, Sub_{-\infty}(\ell < x) = \bot, Sub_{-\infty}(x < u) = \top \tag{2.12}$$

Let $M \models \eta_m$ and $LRAProj_\eta(M) = \eta_m[t]$ where t is either e or $\ell + \epsilon$ or $-\infty$. As η_m is in NNF, it suffices to show that for every literal μ of η_m containing x, the following holds:

$$M \models (\mu \implies Sub_t(\mu)) \tag{2.13}$$

We consider the different possibilities of t below. For a term v, let $M[v]$ denote the value of v in M.

Case $t = e$. In this case, we know that $M \models x = e$. Now, for a literal $\ell < x$,

$$\begin{aligned} M[\ell < x] &\implies M[\ell] < M[x] \\ &= M[\ell] < M[e] \\ &= M[\ell < e] \end{aligned}$$

$$= M[Sub_t(\ell < x)] \qquad \{Sub_t(\ell < x) = (\ell < e)\}.$$

Similarly, literals of the form $x < u$ and $x = e'$ can be considered.

Case $t = \ell + \epsilon$. In this case, we know that $M[\ell < x]$ is true, i.e., $M[\ell] < M[x]$ and whenever $M[\ell' < x]$ is true, $M[\ell' \leq \ell]$ is also true. Now, for a literal $\ell' < x$,

$$\begin{aligned} M[\ell' < x] &\implies M[\ell' \leq \ell] \\ &= M[Sub_t(\ell' < x)] \qquad \{Sub_t(\ell' < x) = (\ell' \leq \ell)\}. \end{aligned}$$

For a literal $x < u$,

$$\begin{aligned} M[x < u] &\implies M[x] < M[u] \\ &\implies M[\ell] < M[u] \qquad \{M[\ell] < M[x]\} \\ &\implies M[\ell < u] \\ &= M[Sub_t(x < u)] \qquad \{Sub_t(x < u) = (\ell < u)\} \end{aligned}$$

For a literal $x = e$, (2.13) vacuously holds as $M[x = e]$ is false.

Case $t = -\infty$. In this case, we know that $M[x = e]$ and $M[\ell < x]$ are false for every literal of the form $x = e$ and $\ell < x$. So, for such literals (2.13) vacuously holds. For a literal $x < u$, $Sub_t(x < u) = \top$ and hence, (2.13) holds again.

□

2.5.3 MBP for Linear Integer Arithmetic (LIA)

We will now present our MBP $LIAProj_\eta$ for LIA. It is based on Cooper's method for Quantifier Elimination procedure for LIA [42]. Let $\eta(\overline{y}) = \exists x \cdot \eta_m(x, \overline{y})$, where η_m is quantifier free and in negation normal form. Assume that x is of sort `Int` and that Th is LIA, or its combination with Propositional Logic. Without loss of generality, let the only literals containing x be the form $\ell < x$, $x < u$, $x = e$ or $(d \mid \pm x + w)$, where $a \mid b$ denotes that a divides b, the terms ℓ, u, e and w are x-free, and $d \in \mathbb{Z} \setminus \{0\}$. Let $E = \{e \mid (x = e) \in lits(\eta_m)\}$ be the set of equality terms of x and $L = \{\ell \mid (\ell < x) \in lits(\eta_m)\}$ be the set of lower-bounds of x. Then, by Cooper's method,

$$\exists x \cdot \eta_m(x, \overline{y}) \equiv \bigvee_{(x=e) \in lits(\eta)} \eta_m[e] \vee \bigvee_{(\ell < x) \in lits(\eta)} \left(\bigvee_{i=0}^{D-1} \eta_m[\ell + 1 + i] \right) \vee \bigvee_{i=0}^{D-1} \eta_m^{-\infty}[i]. \quad (2.14)$$

where D is the least common multiple of all the divisors in the divisibility literals of η_m, $[\cdot]$ denotes a substitution for x and $\eta_m^{-\infty}$ is obtained from η_m by substituting all non-divisibility literals as follows:

$$(\ell < x) \mapsto \bot \qquad (x < u) \mapsto \top \qquad (x = e) \mapsto \bot \quad (2.15)$$

Intuitively, the disjunction partitions the space of the possible values of x. A disjunct for $(x = e)$ covers the case when x is equal to an equality term. The remaining disjuncts cover the cases where ℓ is the maximal lower bound of x and where x satisfies no lower bound. The disjunction over the possible values of i covers

the different ways in which the divisibility literals can be satisfied.

Model based projection $LIAProj_\eta$ is defined as follows, conflicts are resolved by some arbitrary, but fixed, syntactic ordering on terms:

$$LIAProj_\eta(M) = \begin{cases} \eta_m[e], & \text{if } x = e \in lits(\eta) \wedge M \models (x = e) \\ \eta_m[\ell + 1 + i_\ell], & \text{else if } (\ell < x) \in lits(\eta) \wedge M \models (\ell < x) \wedge \\ & \forall (\ell' < x) \in lits(\eta) \cdot M \models ((\ell' < x) \implies (\ell' \leq \ell)) \\ \eta_m^{-\infty}[i_{-\infty}], & \text{otherwise} \end{cases} \tag{2.16}$$

where $i_\ell = M[x - (\ell + 1)] \bmod D$, $i_{-\infty} = M[x] \bmod D$, and $M[x]$ is the value of x in M.

The following theorem shows that $LIAProj_\eta$ is indeed a model based projection. The proof is similar to that of Theorem 5.

Theorem 6. *$LIAProj_\eta$ is a Model Based Projection.*

2.5.4 Bounded Safety with MBP

Given an MBP $Proj_\eta$ for an existentially quantified formula η, we have seen above that each quantifier-free formula in the image of $Proj_\eta$ under-approximates η. As above, we use η_m for the quantifier-free matrix of η. We can now modify the side-condition $\psi = \eta$ of MUST and QUERY in the algorithm BNDSAFETY to use quantifier-free under-approximations as follows: (i) for MUST, the new side-condition is $\psi = Proj_\eta(M)$ where $M \models \eta_m \wedge \varphi$, and (ii) for QUERY, the new side-condition is $\psi = Proj_\eta(M)$ where $M \models \eta_m \wedge [\![\Sigma_R(a)]\!]_o^{b-1}$. Note that to avoid redundant applications of

the rules, we require M to satisfy a formula stronger than η_m. Intuitively, (i) ensures that the newly inferred reachability fact answers the current query and (ii) ensures that the new query cannot be immediately answered by known facts. In both cases, the required model M can be obtained as a side-effect of discharging the premises of the rules. Soundness of BndSafety is unaffected and termination of BndSafety follows from the image-finiteness of $Proj_\eta$.

Theorem 7. *Assuming an oracle and an MBP for Th,* BndSafety *is sound and terminating after modifying the rules as described above.*

Proof. Here, we show that BndSafety with MBP is sound and terminating.

First of all, in presence of MBP, May is unaffected and a reachability fact inferred by Must is only strengthened. Thus, soundness of BndSafety (Theorem 1) is preserved.

Then, it is easy to show that the modified side-conditions to Must and Query preserve Lemmas 1 and 2 and we skip the proof.

Then, we will show that the *finite-image* property of an MBP preserves the finiteness of the number of reachability facts inferred and the number of queries generated by the algorithm. Let d be the size of the image of an MBP. In the proof of Lemma 3, the recurrence relation will now have an extra factor of d. The rest of the proof of finiteness of the number of reachability facts remains the same. Similarly, in the proof of Lemma 4, the number of times Query can be applied along a path for a fixed division and fixed environments O_σ^{b-1} and U_ρ^{b-1} will increase by a factor of d. Again, the rest of the proof of finiteness of the number of queries generated remains the same. That is, Lemmas 3 and 4, and hence, Lemma 6, are preserved

with scaled up complexity bounds.

Note that Theorem 5 is unaffected by under-approximations.

Together, we have that Theorem 2 is preserved, with a scaled up complexity bound. □

Thus, BndSafety with a linear-time MBP (such as $LRAProj_\eta$) keeps the size of the formulas small by efficiently inferring only the necessary under-approximations of the quantified formulas.

2.6 Implementation and Experiments

We have implemented RecMC for analyzing C programs as part of our tool Spacer. The back-end is based on Z3 [45] which is used for SMT-solving and interpolation. It supports propositional logic, linear arithmetic, and bit-vectors (via bit-blasting). The front-end is based on the tool UFO [8]. It encodes safety of a C program by converting it to the Horn-SMT format of Z3, which corresponds to the logical program representation described in Section 2.3. Loops are handled by creating fresh predicate symbols denoting the loop invariants and encoding the corresponding verification conditions. The implementation and benchmarks are available online[3]. We evaluated Spacer on three sets of benchmarks:

(a) 2,908 Boolean programs obtained from the SLAM toolkit, [4]

(b) 1,535 procedural programs from Microsoft's SDV project, [5] and

[3] http://www.cs.cmu.edu/~akomurav/projects/spacer/home.html.
[4] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/BOOL/slam.zip
[5] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/ALIA/sdv

(c) 797 C programs from the Software Verification Competition (SV-COMP) 2014 [4]

The numbers of programs mentioned for the second and third sets of benchmarks above exclude programs with memory-related properties as SPACER cannot handle them yet. The 797 programs in the third set of benchmarks also exclude programs that can be easily verified by our front-end (which converts a C program to the Horn-SMT format) using common compiler optimizations. Note that the programs in the last set are not recursive and our current front-end inlines all procedure calls. We call the resulting set of encodings SVCOMP-1. Note that SVCOMP-1 essentially corresponds to while-programs. We introduced procedural modularity in SVCOMP-1 by two distinct means: (a) factoring out maximal loop-free fragments into new loop-free, recursion-free procedures (the main procedure may still have loops) to obtain SVCOMP-2, and (b) factoring out loops into tail-recursive procedures (in an inside-out fashion for nested loops) to obtain SVCOMP-3. Our simple outlining procedure could not handle some large programs and SVCOMP-3 has 45 fewer programs.

Fig. 2.8 shows some characteristics of the Horn-SMT encodings for the benchmarks, when viewed according to the logical program representation described in Section 2.3. In particular, for SVCOMP-1, we consider a tail-recursive view of the inlined encodings. The number of calls along a path roughly identifies the procedural modularity of the encodings. The number of calls of a procedure (in the entire program) identifies the potential number of times a summary (may or must) of the procedure can be reused for a given bound on the call-stack. Note that summaries can also be reused across different bounds on the call-stack. Despite the fact that the average number of procedure calls is low from the figure, we can show the practical

	#calls along a path		#calls of a procedure	
	max (over all paths)	*avg* (over all paths)	*max* (over all procedures)	*avg* (over all procedures)
SLAM	2	1	95.1	1.4
SDV	11.9	1.18	21.6	1.7
SVCOMP-1	1	0.7	4.1	1.4
SVCOMP-2	2	0.8	3.9	1.1
SVCOMP-3	1.5	0.8	8.4	3.2

Figure 2.8: Some characteristics of the Horn-SMT encodings of the benchmarks, averaged over all programs in the corresponding set.

advantage of RECMC using these benchmarks, as we show below.

We compared SPACER against the implementation of GPDR in Z3 [74]. GPDR is inspired by the IC3 hardware model checking algorithm [25] and avoids unrolling the call-graph. Thus, it creates and checks reachability queries for individual procedures similar to RECMC. However, it only computes may summaries and because of the lack of must summaries, its query creation mechanism is quite different from the rule QUERY. Moreover, it does not use MBP.

In our experiments, the resource limits were set to 30 minutes of time and 16GB of memory, on an Ubuntu machine with a 2.2 GHz AMD Opteron(TM) Processor 6174 and 516GB RAM. Fig. 2.9 and 2.10 show a high level summary of the results in terms of the number of programs verified by SPACER and Z3. Since there are some programs verified by only one of the tools, the figures also report the number of programs verified by at least one tool in the third row. We provide a more detailed discussion of the experimental results in the following. In the scatter plots that are shown below, a diamond indicates a time-out and a star indicates a mem-out.

Boolean Program Benchmarks. Fig. 2.11(a) shows the scatter plot of runtimes for SPACER and Z3 for the SLAM benchmarks. The runtimes of both the tools are within ± 5 minutes for over 98% of the benchmarks. Of the remaining, SPACER is

	Slam		Sdv	
	SAFE	UNSAFE	SAFE	UNSAFE
Spacer	1,721	985	1,303	232
Z3	1,727	992	1,302	232
Spacer or Z3	1,727	992	1,303	232

Figure 2.9: Number of programs verified for Slam and Sdv benchmarks.

	Svcomp-1		Svcomp-2		Svcomp-3	
	SAFE	UNSAFE	SAFE	UNSAFE	SAFE	UNSAFE
Spacer	249	509	213	497	234	482
Z3	245	509	208	493	234	477
Spacer or Z3	252	509	225	500	240	482

Figure 2.10: Number of programs verified for SVCOMP benchmarks.

better on 1 benchmark, Z3 is better on 42 benchmarks which includes 13 benchmarks where Spacer runs out of time. Recall that Z3 utilizes may summaries which heuristically avoid the possible exponential blow-up associated with unwinding the call-graph and as the plot shows, such a heuristic approach can be better than Spacer in some cases. However, when we compared the tools on the parametric Boolean program from Fig. 1.1, in which the size of the unrolled call-tree necessarily grows exponentially in the number of procedures, Spacer handles the increasing complexity significantly better than Z3, as shown in Fig. 2.11(b).

SDV Benchmarks. Fig. 2.12 shows the scatter plot of runtimes for Spacer and Z3 for the SDV benchmarks. Spacer clearly outperforms Z3 including a benchmark where Z3 runs out of time.

SVCOMP 2014 Benchmarks. We begin with the scatter plot in Fig. 2.13(a) for Svcomp-1 benchmarks. As mentioned above, Svcomp-1 benchmarks correspond to while-programs and therefore, do not require must summaries. As the GPDR algorithm also computes may summaries, the plot in Fig. 2.13(a) essentially shows the advantage of using MBP in creating a new query as opposed to Z3's variable

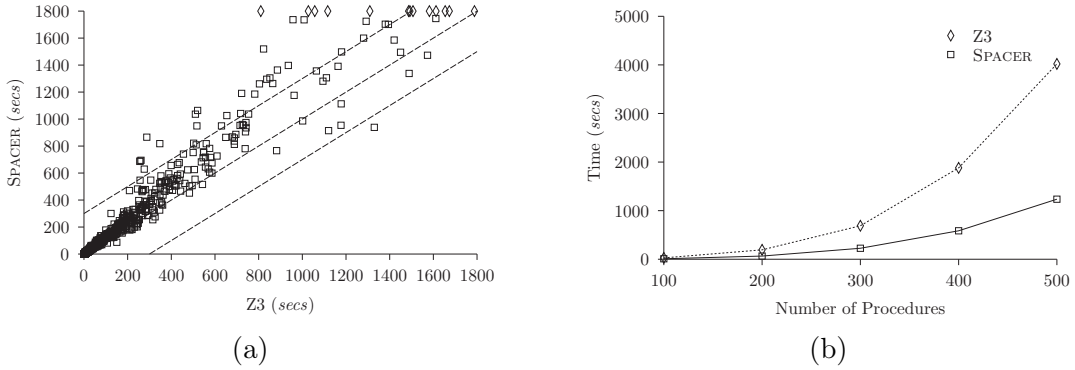


Figure 2.11: SPACER vs. Z3 for (a) the SLAM benchmarks (with ±5 minute boundaries), and (b) the Boolean program in Fig. 1.1 which is parametric in the number of procedures.

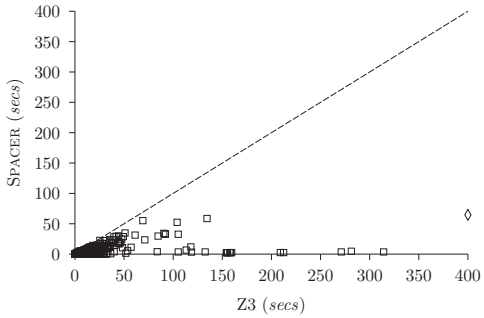


Figure 2.12: SPACER vs. Z3 for the SDV benchmarks.

substitution based on a given model.[6]

To understand the effect of must summaries, we also created a version of SPACER that only infers and utilizes may summaries. We obtained this by modifying Z3 to use MBP in creating new queries. As shown in Fig. 2.13(b), the advantage of using must summaries is quite significant on SVCOMP-2 benchmarks.

So, a combination of MBP and must summaries is expected to result in significant improvements over using may summaries alone. This is shown experimentally in Fig. 2.14(a) and 2.14(b) for the SVCOMP-2 and SVCOMP-3 benchmarks which show

[6]Z3 first tries to eliminate existential quantifiers by using equalities with ground terms present in the input formula and resorts to model substitution otherwise.

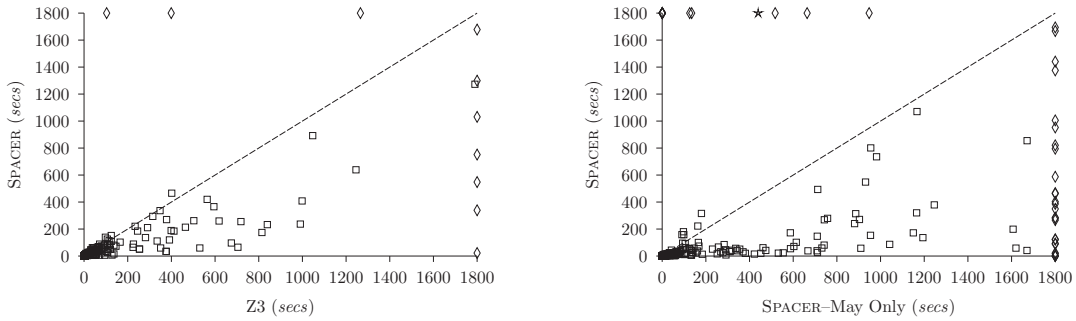


Figure 2.13: The advantage of (a) MBP, over SVCOMP-1, and (b) must summaries, over SVCOMP-2, in SPACER. For SVCOMP-1, must summaries are not required and MBP is the only key difference between SPACER and Z3.

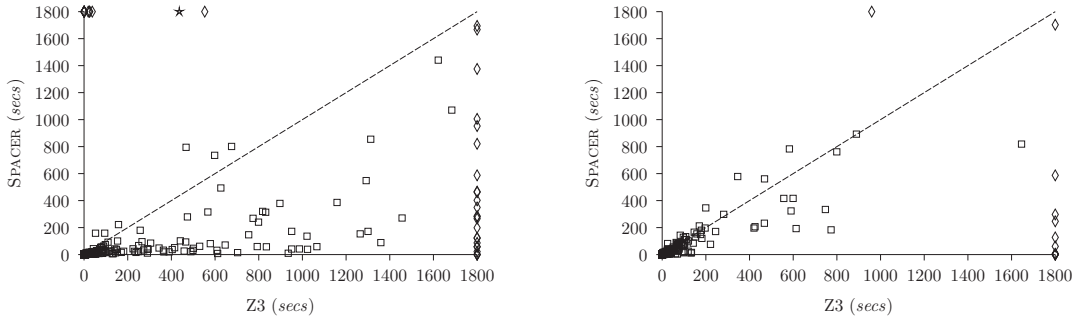
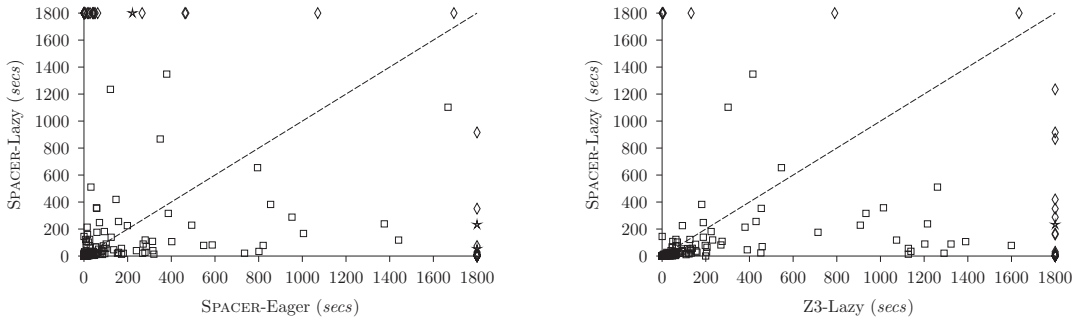


Figure 2.14: SPACER vs. Z3 for the benchmarks (a) SVCOMP-2 and (b) SVCOMP-3.

that SPACER is significantly better than Z3 on most of the programs.

Recall that the rule QUERY checks the feasibility of a potential counterexample path π by recursively creating a new reachability query for a procedure R called along π. Due to our logical representation of a program, one can consider an arbitrary permutation of the conjuncts of π when applying the rule and the choice of the procedure R is not deterministic. Our current implementation in SPACER can order the conjuncts either in the given order or in the reversed order and for lack of good heuristics, we do not consider other permutations. These two orderings correspond

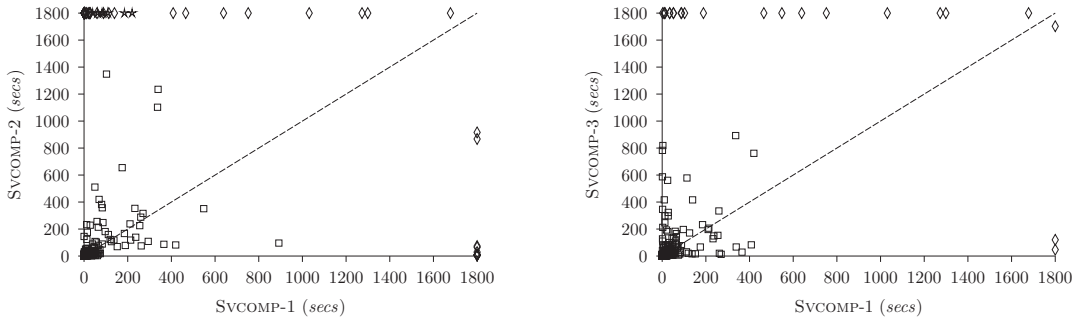


(a) (b)

Figure 2.15: Effect of the order of query creation in QUERY in SPACER and the corresponding order of query handling in Z3, on SVCOMP-2 benchmarks.

to top-down and bottom-up feasibility analyses. In particular, the plot shown in Fig. 2.14(a) corresponds to a bottom-up analysis.

As mentioned in the beginning, the SVCOMP-2 benchmarks are obtained by taking the while-program encodings in SVCOMP-1 and factoring out maximal loop-free fragments into new loop-free, recursion-free procedures. Furthermore, as also mentioned in the beginning, loops are encoded in SVCOMP-1 by introducing new predicate symbols that denote loop invariants and by encoding the corresponding verification conditions. So, a path in a procedure in the resulting logical encoding (see Section 2.3) contains at most two calls, one corresponding to an invariant at a control location and the other corresponding to a newly introduced procedure for a loop-free fragment. Thus, a top-down analysis refines the may summaries of the new procedures only when necessary, similar to a CEGAR-style reasoning where the may summaries of the new procedures abstract the loop-free fragments. We call this a *lazy* refinement strategy. In contrast, a bottom-up analysis on these benchmarks corresponds to an *eager* refinement strategy which is shown in Fig. 2.14(a).



(a) (b)

Figure 2.16: Comparison of SPACER's behavior on the various encodings of SVCOMP benchmarks. For the plot in (a), we use SPACER in the *lazy* mode for SVCOMP-2.

Fig. 2.15(a) shows a scatter plot of runtimes on SVCOMP-2 comparing the behavior of SPACER for the two orderings. While it is unclear from the figure which ordering is better, SPACER continues to outperform Z3 even with the lazy strategy, as shown in Fig. 2.15(b).

Finally, as an interesting exercise, we compared the behavior of SPACER on various encodings of the SVCOMP benchmarks. For the comparison of runtimes between SVCOMP-2 and SVCOMP-1, we considered the lazy mode of SPACER for the SVCOMP-2 encodings, which essentially corresponds to abstract reasoning by inferring sufficient summaries of the loop-free fragments. Fig. 2.16(a) shows the runtime comparison for the benchmarks. While the SVCOMP-2 encodings seem to be worse overall, the difference in performance between the encodings is less clear when we restrict ourselves to the harder benchmarks, e.g., where the SVCOMP-1 encodings need more than 5 minutes of runtime. However, as we will see in Chapter 3, abstraction can be quite powerful and we plan to incorporate the ideas from that chapter into the framework of RECMC in the future. Then, Fig. 2.16(b) shows the runtime

comparison for the encoding SVCOMP-3 against SVCOMP-1. Recall that SVCOMP-3 encodings are obtained by factoring out loops into tail-recursive procedures. In other words, we are replacing the inference of loop invariants by that of summaries of the corresponding tail-recursive procedures. Whereas a loop invariant depends on the variables in scope, the signature of the corresponding tail-recursive procedure, and hence its summary, depends on two copies of the variables in scope which denote their values before and after a loop iteration. As the plot shows, this can negatively affect the performance of verification.

Overall, we have shown significant practical benefits of the core ideas behind RECMC using our implementation in SPACER and various realistic benchmarks.

2.7 Related Work

There is a large body of work on interprocedural program analysis. It was pointed out early on that safety verification of recursive programs is reducible to the computation of a fixed-point over relations representing the input-output behavior of each procedure [37]. The term summary is used for such a relation in the *functional approach* of Sharir and Pnueli [103]. Reps, Horwitz, and Sagiv [101] showed that for a large class of finite, interprocedural dataflow problems, the summaries can be computed in time polynomial in the number of dataflow facts and procedures. Ball and Rajamani [19] adapted the RHS algorithm to the verification of Boolean Programs as part of the SLAM project for software model checking using a CEGAR-style loop with predicate abstraction [62]. Following SLAM, other software model checkers, e.g., BLAST [71]

and MAGIC [30], also implemented predicate abstraction based algorithms. While predicate abstraction is used to obtain over-approximations of procedure semantics, these approaches do not use under-approximations as we do.

In the context of predicate abstraction, the algorithm SMASH also combines over- and under-approximations for analyzing procedural programs [60]. However, the summaries in SMASH can have auxiliary variables which differ from one calling context to another, restricting the reusability of the summaries. SMASH also under-approximates existential quantification in computing the results of the *post* and *pre* operations, but unlike RECMC, the under-approximations are obtained using concrete values encountered during testing of the program.

As mentioned earlier in the paper, several SMT-based algorithms have been proposed for safety verification of recursive programs, including WHALE [7], HSF [63], Duality [92], Ultimate Automizer [68, 69], and Corral [84]. These algorithms share a similar structure – they use SMT-solvers to look for counterexamples and interpolation to compute over-approximating procedure summaries. The algorithms differ in the SMT encoding and the heuristics used. However, in the worst-case, they completely unroll the call graph into a tree.

The work closest to ours is GPDR [74], which extends the hardware model checking algorithm IC3 of Bradley [25] to SMT-supported theories and recursive programs. Unlike RECMC, GPDR does not maintain must-summaries. In the context of Fig. 2.6, this means that σ_u is always empty and there is no MUST rule. Instead, the QUERY rule is modified to use a model M that satisfies the premises (instead of our use of the entire path π when creating a query). Furthermore, undesirable

reachable states are cached. While the algorithm terminates for Boolean programs, a formula can have infinitely many models in the general case of first-order languages and GPDR might end up applying the QUERY rule indefinitely (see Appendix 2.A). In contrast, RECMC creates only finitely many queries for a given bound on the call-stack depth and is guaranteed to find a counterexample if one exists.

In the context of Boolean programs, there also exists a SAT-based summarization technique that allows extra *choice variables* in the formulas and thereby requires a Quantified Boolean Formulas (QBF) solver to check for convergence [20].

2.8 Conclusion

We presented RECMC, a new SMT-based algorithm for model checking safety properties of recursive programs. For programs and properties over decidable theories, RECMC is guaranteed to find a counterexample if one exists. To our knowledge, this is the first SMT-based algorithm with such a guarantee while being polynomial for Boolean Programs. The key idea is to use a combination of under- and over-approximations of the semantics of procedures, avoiding re-exploration of parts of the state-space. We described an efficient instantiation of RECMC for Linear Arithmetic (over rationals and integers) by introducing *Model Based Projection* to under-approximate the expensive quantifier elimination. We have implemented it in our tool SPACER and shown empirical evidence that it significantly improves on the state-of-the-art.

In the future, we would like to explore extensions to other theories. Of particular

interest are the theory EUF of uninterpreted functions with equality and the theory of arrays. The challenge is to deal with the lack of quantifier elimination. Another direction of interest is to combine RecMC with *Proof-based Abstraction* [66, 80, 91], which also forms the basis of the next chapter, to explore a combination of the approximations of procedure semantics with transition-relation abstraction.

The algorithm RecMC and the results presented in this chapter are published as part of the proceedings of CAV 2014 [81].

2.A Divergence of GPDR for Bounded Call-Stack

Consider the program $\langle\langle M, L, G\rangle, M\rangle$ with procedures $M = \langle y_0, y, \Sigma_M, \langle x, n\rangle, \beta_M\rangle$, $L = \langle n, \langle x, y, i\rangle, \Sigma_L, \langle x_0, y_0, i_0\rangle, \beta_L\rangle$, and $G = \langle x_0, x_1, \Sigma_G, \emptyset, \beta_G\rangle$ where:

$$\beta_M = \Sigma_L(x, y_0, n, n) \wedge \Sigma_G(x, y) \wedge n > 0$$

$$\beta_L = (i = 0 \wedge x = 0 \wedge y = 0) \vee$$

$$(\Sigma_L(x_0, y_0, i_0, n) \wedge x = x_0 + 1 \wedge y = y_0 + 1 \wedge i = i_0 + 1 \wedge i > 0)$$

$$\beta_G = (x = x_0 + 1)$$

The GPDR [74] algorithm can be shown to diverge when checking the bounded safety problem $M \models_2 y_0 \leq y$, for e.g., by inferring the diverging sequence of over-approximations of $[\![L]\!]^1$: $(x < 2 \implies y \leq 1), (x < 3 \implies y \leq 2), \ldots$.

We also observed this behavior experimentally (Z3 revision `d548c51` at `http://z3.codeplex.com`).[7]

[7] Horn-SMT file: `http://www.cs.cmu.edu/~akomurav/projects/spacer/gpdr_diverging.smt2`.

Chapter 3

Abstraction in SMT-Based Model Checking

3.1 Introduction

As described in Chapter 1, SMT-based model checkers work by deciding bounded safety for increasing values of the bound on the length of an execution. When the safety property holds, the termination of such algorithms in practice depends on whether a proof of bounded safety can be found that also proves (unbounded) safety. Not surprisingly, given the undecidability of safety, this can be quite challenging to achieve in practice. In this chapter, we present SPACER[1], an algorithm that incorporates *automatic abstraction refinement* into SMT-based model checking.

Consider the safe program P_g (adapted from [65]) shown in Fig. 3.1. Here,

[1]Software Proof-based Abstraction with CounterExample-based Refinement.

```
0: x=0; y=0; z=0; w=0;
1: while (nd_bool()) {
2:   if (nd_bool()) {x++; y=y+100;}
3:   else if (nd_bool())
4:     if (x>=4) {x++; y++;}
5:   else if (y>10*w && z>=100*x) {y=-y;}
6:   t=1;
7:   w=w+t; z=z+(10*t);
   }
8: assert(!(x>=4 && y<=2));
```

Figure 3.1: A program P_g adapted from an example by Gulavani et al. [65].

`nd_bool` is a routine that returns a Boolean value.[2] P_g is hard for existing SMT-based algorithms. For example, the implementation of the algorithm GPDR[3] [74] in Z3 [45] (v4.3.1) cannot verify the program in an hour. However, an *abstraction* of the program, $\hat{P}_g$, obtained by replacing line 6 with a non-deterministic assignment to `t` is verified by the same tool in under a second. Our implementation of Spacer finds a safe abstraction of P_g in under a minute (the transition relation of the abstraction we automatically computed is a non-trivial generalization of that of P_g and does not correspond to $\hat{P}_g$).

The key intuition behind Spacer is that a *good* abstraction of the program can lead to a *good* proof of bounded safety. That is, the assertions in a proof of bounded safety that over-approximate the reachable states at the top of a loop or the behavior of a procedure can be less dependent on the bound, because of the abstraction. This can, in turn, help in faster convergence to inferring invariants in a fewer number of iterations of bounded safety. As a proof does not utilize all the details of the program, in general, Spacer obtains a program abstraction by hiding the details of

[2]In other words, assume that the behavior of `nd_bool` is unknown. So, for the purpose of verification, `nd_bool` effectively returns either `true` or `false` non-deterministically.

[3]GPDR stands for *Generalized Property Directed Reachability*.

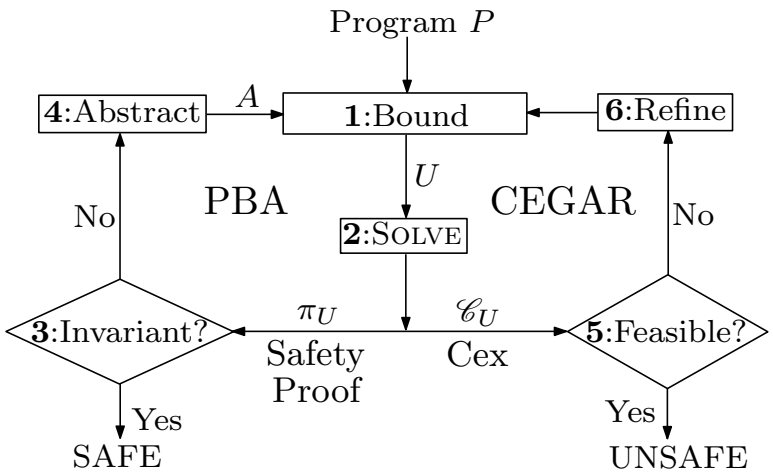


Figure 3.2: An overview of SPACER.

the transition relation irrelevant for a proof of bounded safety (called *Proof-Based Abstraction* [66, 91] (PBA)). However, an abstraction can be too coarse in which case we utilize spurious abstract counterexamples for refinement (called *CounterExample Guided Abstraction Refinement* [38] (CEGAR)).

Fig. 3.2 shows the high level flow of our algorithm SPACER. We assume that the input program P is annotated with the given safety property (e.g., using `assert` statements). SPACER begins with an initial abstraction A of P (which can be P itself). Each iteration of SPACER starts by obtaining an under-approximation U of A and checking safety of U (steps **1** and **2**). The under-approximations we will consider in this chapter are obtained by bounding the length of an execution. If U is safe, we obtain a proof π_U (as invariants), and otherwise, we obtain a counterexample to safety $\mathscr{C}_U$. In practice, the safety check is implemented using an interpolating SMT-solver (e.g., [45, 64, 77]) or a *generalized Horn-Clause* solver (e.g., [63, 74, 92], including the algorithm RECMC described in Chapter 2). If U is proved safe, it is

first checked whether the formulas in π_U are also invariant for the original program P, in which case SPACER outputs SAFE (step **3**); otherwise, a new abstraction of P is obtained using the proof π_U (step **4**; see below for details) and the next iteration begins. On the other hand, if U is proved unsafe, $\mathscr{C}_U$ is an abstract counterexample and needs to be checked for feasibility in P (step **5**; this is based on the well-known CEGAR approach [38]). If $\mathscr{C}_U$ is feasible, SPACER outputs UNSAFE; otherwise, the abstraction A is refined to eliminate the spurious counterexample (step **6**) and the next iteration begins. SPACER is described in Section 3.4 and a detailed run of the algorithm on an example is given in Section 3.2.

Note that the left iteration of SPACER (steps **1**–**4**) is PBA: in each iteration, an under-approximation is verified, a new abstraction based on the proof is computed and a new under-approximation is constructed. To the best of our knowledge, this is the first application of PBA to Software Model Checking. The right iteration (steps **1**, **2**, **5**, **6**) is CEGAR: in each iteration, (an under-approximation of) an abstraction is verified and refined by eliminating spurious counterexamples. SPACER exploits the natural duality between the two.

We have implemented SPACER using the GPDR engine inside the tool Z3 [74] as SOLVE (see Section 3.5) and evaluated it on many benchmarks from the 2nd Software Verification Competition[4] (SV-COMP'13). Our experimental results (see Section 3.6) show that abstraction significantly improves the performance of SMT-based model checking on hard benchmarks.

In summary, we present: (a) a new algorithm SPACER that combines abstraction

[4] http://sv-comp.sosy-lab.org

with SMT-based model checking and tightly connects proof- and counterexample-based abstraction-refinement, (b) an implementation of SPACER using Z3, and (c) experimental results showing the effectiveness of SPACER.

3.2 Overview

In this chapter, we restrict ourselves to programs that can be represented by transition systems.[5] Let P be a program represented by the transition system $\langle \overline{v}, \iota(\overline{v}), \tau(\overline{v}, \overline{v}'), \mathit{err}(\overline{v})\rangle$, where $\overline{v}$ is the list of state variables and ι, τ, and err denote the initial condition, the transition relation, and the error condition, respectively. Note that we use primed variables to denote the next-state values. Below, we give a brief explanation of our abstraction mechanism based on the proofs (of under-approximations) obtained in an iteration of SPACER.

Proof-Based Abstraction. Given P and a proof π of a property of P (for e.g., π is a proof of *bounded* safety of P for some bound on the possible executions), the goal of *Proof-Based Abstraction* (PBA) is to obtain a program $\hat{P}$ such that

1. $\hat{P}$ is an abstraction of P, i.e., $P \preceq \hat{P}$, and
2. π proves the property for $\hat{P}$.

Here, $\preceq$ denotes the usual simulation conformance between transition systems. Intuitively, when π proves safety of P for a given bound b on the length of an execution, such a proof *preserving* abstraction mechanism ensures that π continues to prove

[5] In particular, this disallows procedure calls. In principle, the ideas presented here can be combined with those in Chapter 2 to handle procedural programs as well, but we leave it for future exploration.

bounded safety for $\hat{P}$. Thus, for the future iterations of SPACER, we can use $\hat{P}$, instead of P, with the hope of getting proofs of bounded safety that depend less on the bound and can lead to faster convergence to inferring invariants. When the abstraction is too coarse, we use the well-known CEGAR approach [38] for refinement.

Our notion of proof in PBA is slightly different from a *refutation proof* given by a SAT solver used in the context of hardware verification [91]. As we will see in Section 3.2.1, each iteration of SPACER checks safety of a different program and hence, our proofs correspond to program invariants.

In particular, for a bound $b \geq 0$ on the length of an execution (i.e., number of transitions) and a fresh program variable c denoting a down-counter for the number of transitions, a formula $\pi(\overline{v}, c)$ is a *proof of bounded safety of P for b* iff the following are valid:

$$\iota(\overline{v}) \implies \pi(\overline{v}, c)$$

$$\pi(\overline{v}, c) \wedge 0 < c \leq b \wedge \tau(\overline{v}, \overline{v}') \wedge c' = c - 1 \implies \pi(\overline{v}', c')$$

$$\pi(\overline{v}, c) \wedge c = 0 \wedge err(\overline{v}) \implies \bot$$

In words, π holds initially and for all states reachable in at most b-many transitions such that it proves safety. Intuitively, π is a *bounded invariant* for the bound b.

Given b and π, one possibility for PBA is to obtain a new program $\hat{P} = \langle \overline{v}, \hat{\iota}(\overline{v}), \hat{\tau}(\overline{v}, \overline{v}'), \hat{err}(\overline{v}) \rangle$ such that

1. $\iota \implies \hat{\iota}$, $\tau \implies \hat{\tau}$, and $err \implies \hat{err}$ are valid, and
2. π proves bounded safety of $\hat{P}$ for b.

In words, the initial condition, the transition relation and the error condition are weakened such that $P \preceq \hat{P}$ and moreover, the proof π of bounded safety is preserved.

However, to obtain more precise abstractions, we perform PBA *relative to known invariants* of P. An invariant is a formula inv such that the following are valid:

$$\iota(\overline{v}) \implies inv(\overline{v}) \tag{3.1}$$

$$inv(\overline{v}) \wedge \tau(\overline{v}, \overline{v}') \implies inv(\overline{v}'). \tag{3.2}$$

In other words, inv holds of every reachable state of P. Note that an invariant need not be safe, i.e., $inv(\overline{v}) \wedge err(\overline{v})$ may be satisfiable.

Given an invariant $inv(\overline{v})$ of P, the goal of PBA relative to inv is to obtain $\hat{P} = \langle \overline{v}, \hat{\iota}(\overline{v}) \wedge inv(\overline{v}), \hat{\tau}(\overline{v}, \overline{v}') \wedge inv(\overline{v}) \wedge inv(\overline{v}'), \hat{err}(\overline{v}) \wedge inv(\overline{v}) \rangle$, where, as before,

1. $\iota \implies \hat{\iota}$, $\tau \implies \hat{\tau}$, and $err \implies \hat{err}$ are valid, and
2. π proves bounded safety of $\hat{P}$ for b.

In words, we combine the weakening of ι, τ, and err with the invariants on the current and the next-state variables. One can easily show that $P \preceq \hat{P}$. Using invariants of P in PBA yields a more precise abstraction because the reachable states of the abstraction are confined to the known invariants.

3.2.1 Example

Consider the example transition system in Fig. 3.3. As described above, the variables b and c in the figure denote the bound on the length of an execution and a fresh variable denoting a down-counter for the number of transitions along an execution,

$$
\begin{array}{rcl}
\overline{v} & = & \langle x, y, z, w \rangle \\
\iota(\overline{v}) & \equiv & (x = y = z = w = 0) \\
\tau(\overline{v}, \overline{v}') & \equiv & [(x' = x + 1 \ \wedge \ y' = y + 100) \vee \\
 & & (x \geq 4 \ \wedge \ x' = x + 1 \ \wedge \ y' = y + 1) \vee \\
 & & (y > 10w \ \wedge \ z \geq 100x \ \wedge \\
 & & \quad y' = -y \ \wedge \ x' = x)] \wedge \\
 & & w' = w + 1 \ \wedge \ z' = z + 10 \ \wedge \\
 & & 0 < c \leq b \ \wedge \ c' = c - 1 \\
err(\overline{v}) & \equiv & c = 0 \ \wedge \ x \geq 4 \ \wedge \ y \leq 2
\end{array}
$$

Figure 3.3: An example program P represented as a transition system.

respectively. Fixing a value of b results in an under-approximation of the program. For example, adding the constraint $b = 0$ to τ in the figure corresponds to the under-approximation that allows no transitions. On the other hand, adding the constraint $b = 1$ instead corresponds to the under-approximation that allows at most one transition. While in this example the variables b and c are part of P, we synthesize such variables automatically in practice (see Section 3.5). In the following, we illustrate SPACER using this example.

Bound and Solve. For $b = 2$, one possible proof of bounded safety, say π_2, is obtained as the conjunction of the set of clauses shown in Fig. 3.5(a).

Extract Invariants. The next step of SPACER is to check if π_2 also proves unbounded safety of P. For this purpose, we first obtain a *maximal subset* $\mathcal{I}_2$ of the set of clauses of π_2 that are invariant for P, i.e., $inv = \bigwedge \mathcal{I}_2$ makes the two formulas (3.1) and (3.2) valid. We call such a subset $\mathcal{I}_2$ a *Maximal Inductive Subset* (MIS) of π_2. Fig. 3.5(b) shows one such subset $\mathcal{I}_2$. In this case, $\mathcal{I}_2$ does not prove safety, i.e., $\bigwedge \mathcal{I}_2$ is satisfiable with err. Nevertheless, as a by-product of this step, we have obtained non-trivial invariants $\mathcal{I}_2$ of P.

$$
\begin{array}{rcl}
\overline{v} & = & \langle x, y, z, w \rangle \\
\hat{\iota}_1(\overline{v}) & \equiv & (x = y = z = w = 0) \wedge \\
 & & inv(x, y, z, w) \\
\hat{\tau}_1(\overline{v}, \overline{v}') & \equiv & [(x' = x + 1) \vee \\
 & & (x \geq 4 \ \wedge \ x' = x + 1) \vee \\
 & & (y > 10w \ \wedge \ z \geq 100x)] \wedge \\
 & & 0 < c \leq b \ \wedge \ c' = c - 1 \ \wedge \\
 & & inv(x, y, z, w, c) \wedge inv(x', y', z', w', c') \\
e\hat{r}r_1(\overline{v}) & \equiv & c = 0 \ \wedge \ x \geq 4 \wedge \\
 & & inv(x, y, z, w, c)
\end{array}
$$

(a) $\hat{P}_1$

$$
\begin{array}{rcl}
\overline{v} & = & \langle x, y, z, w \rangle \\
\hat{\iota}_2(\overline{v}) & \equiv & (x = y = z = w = 0) \wedge \\
 & & inv(x, y, z, w) \\
\hat{\tau}_2(\overline{v}, \overline{v}') & \equiv & [(x' = x + 1 \ \wedge \ y' = y + 100) \vee \\
 & & (x \geq 4 \ \wedge \ x' = x + 1 \ \wedge \ y' = y + 1) \vee \\
 & & (y > 10w \ \wedge \ z \geq 100x)] \wedge \\
 & & 0 < c \leq b \ \wedge \ c' = c - 1 \ \wedge \\
 & & inv(x, y, z, w, c) \wedge inv(x', y', z', w', c') \\
e\hat{r}r_2(\overline{v}) & \equiv & c = 0 \ \wedge \ x \geq 4 \ \wedge \ y \leq 2 \wedge \\
 & & inv(x, y, z, w, c)
\end{array}
$$

(b) $\hat{P}_2$

Figure 3.4: Abstractions $\hat{P}_1$ and $\hat{P}_2$ of P in Fig. 3.3. *inv* denotes $\bigwedge \mathcal{I}_2$ for $\mathcal{I}_2$ in Fig. 3.5(b).

PBA. The next step of SPACER is to perform PBA relative to known invariants $\mathcal{I}_2$. $\hat{P}_1$ in Fig. 3.4(a) is one such abstraction where π_2 continues to be a proof of bounded safety of $\hat{P}_1$ for $b = 2$. In practice, we obtain the abstractions using an *unsatisfiability core* of the SMT problem used to validate the proof of bounded safety. Note that $\hat{\tau}_1$ no longer has the constraints on the next-state values of z, y, and w present in τ as they are captured by the invariant obtained in the previous step. In other words, while $\hat{\tau}_1$ is obtained using a structural (or *syntactic*) abstraction [16], the use of invariants makes it a more expressive, *semantic*, abstraction mechanism.

Bound and Solve. SPACER now modifies the bound constraint to $b = 4$ which results in a counterexample using $\hat{P}_1$. One possible counterexample execution $\mathscr{C}_4$ is shown below which corresponds to incrementing x from 0 to 4:

$$
\begin{array}{l}
\langle (0,0,0,0,0,0), (0,0,0,0,4,4), (1,0,0,0,3,4), (2,0,0,0,2,4), \\
(3,0,0,0,1,4), (4,3,0,0,0,4), (4,3,0,0,0,4) \rangle
\end{array}
\tag{3.3}
$$

where each tuple denotes a valuation of the state variables $\langle x, y, z, w, c, b \rangle$.

Feasibility Check and Refinement. $\mathscr{C}_4$ can be shown to be infeasible in P and

$\{(z \leq 100x - 90 \vee y \leq 10w),$
$z \leq 100x,\ \ x \leq 2,$
$(x \leq 0 \vee c \leq 1),$
$(x \leq 1 \vee c \leq 0)\}$

(a) π_2

$\{(z \leq 100x - 90 \vee y \leq 10w),$
$z \leq 100x\}$

(b) $\mathcal{I}_2$

$\{(z \leq 100x - 90 \vee y \leq 10w),$
$z \leq 100x, y \geq 0,$
$(x \leq 0 \vee y \geq 100)\}$

(c) π_4

Figure 3.5: Proofs and invariants found by SPACER for the program P in Fig. 3.3.

SPACER refines the current abstraction $\hat{P}_1$ to $\hat{P}_2$, shown in Fig. 3.4(b), using CEGAR.

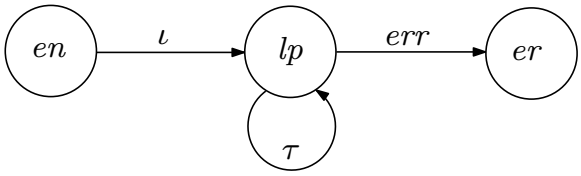
Bound and Solve. SPACER checks bounded safety for $b = 4$ with the new abstraction $\hat{P}_2$ and Fig. 3.5(c) shows one possible proof π_4 as a set of clauses.

Proof-of-Safety Check. One can show that π_4 is MIS of itself, i.e., $\bigwedge \pi_4$ is also an invariant of P and hence, a safety proof of P. At this point, SPACER terminates and outputs SAFE.

While we have carefully chosen the values of the bound to save space, the abstractions, proofs, and invariants shown above were all computed automatically by our implementation starting with the initial bound $b = 0$ and incrementing the bound by 1, each iteration. Even on this small example, abstraction improves the runtime by five times.

3.3 Preliminaries

The transition systems we have seen in Section 3.2 are simplistic and hide the control structure of the input program. In this section, we consider a more general graph representation of a program that explicates the control structure. As mentioned earlier, we restrict ourselves to while-programs. As in Section 2.3, consider a first-



$$
\begin{array}{rcl}
\overline{v} & = & \langle x, y, z, w \rangle \\
\iota(\overline{v}) & \equiv & (x = y = z = w = 0) \\
\tau(\overline{v}, \overline{v}') & \equiv & [(x' = x + 1 \ \wedge \ y' = y + 100) \vee \\
 & & \ (x \geq 4 \ \wedge \ x' = x + 1 \ \wedge \ y' = y + 1) \vee \\
 & & \ (y > 10w \ \wedge \ z \geq 100x \ \wedge \\
 & & \quad y' = -y \ \wedge \ x' = x)] \wedge \\
 & & \ w' = w + 1 \ \wedge \ z' = z + 10 \ \wedge \\
 & & \ 0 < c \leq b \ \wedge \ c' = c - 1 \\
err(\overline{v}) & \equiv & c = 0 \ \wedge \ x \geq 4 \ \wedge \ y \leq 2
\end{array}
$$

Figure 3.6: A graph representation of the transition system in Fig. 3.3.

order language with signature $\mathcal{S}$ and let Th be an $\mathcal{S}$-theory.

Definition 3 (Program)**.** *A program P is a tuple $\langle L, \ell^o, \ell^e, V, \tau \rangle$ where*

1. *L is a finite set denoting the control locations,*
2. *$\ell^o \in L$ and $\ell^e \in L$ are the unique initial and error locations,*
3. *V is a finite set of program variables disjoint from $\mathcal{S}$, and*
4. *τ is a map from pairs of locations in $L \times L$ to quantifier-free sentences over the signature $(\mathcal{S} \cup V \cup V')$, where V' is obtained from V by priming each variable in V and denotes the next-state values of the variables in V.*

Intuitively, $\tau(\ell_i, \ell_j)$ is the relation between the current values of V at ℓ_i and the next values of V at ℓ_j on a transition from ℓ_i to ℓ_j. We refer to τ as the transition relation. Without loss of generality, we assume that there is no incoming transition to the initial location ℓ^o and no outgoing transition from the error location ℓ^e, i.e., for all $\ell \in L$, $\tau(\ell, \ell^o) = \bot$ and $\tau(\ell^e, \ell) = \bot$. We refer to the components of a program P by a subscript, e.g., L_P denotes the set of locations of P.

For example, Fig. 3.6 shows a program $\langle L, \ell^o, \ell^e, V, \tau\rangle$ corresponding to the transition system in Fig. 3.3, where $L = \{en, lp, er\}$, $\ell^o = en$, $\ell^e = er$, $V = \{x, y, z, w, c, b\}$, $\tau(en, lp) = I$, $\tau(lp, lp) = T$, $\tau(lp, er) = E$.

Let $P = \langle L, \ell^o, \ell^e, V, \tau\rangle$ be a program. A *control path* of P is a finite[6] sequence of control locations $\langle \ell^o = \ell_0, \ell_1, \ldots, \ell_k\rangle$, beginning with the initial location ℓ^o, such that $\tau(\ell_i, \ell_{i+1}) \neq \bot$ for $0 \leq i < k$. A *state* of P is an assignment to the variables in V. A control path $\langle \ell^o = \ell_0, \ell_1, \ldots, \ell_k\rangle$ is called *feasible* iff there is an $\mathcal{S}$-structure I with $I \models Th$ and a sequence of states $\langle s_0, s_1, \ldots, s_k\rangle$ such that

$$I\{V \mapsto s_i\}\{V' \mapsto s_{i+1}\} \models \tau(\ell_i, \ell_{i+1}), \text{ for all } 0 \leq i < k \tag{3.4}$$

i.e., the sequence of states is an execution along the control path (the reader is referred to Section 2.3 for the notation used above).

For example, $\langle en, lp, lp, lp\rangle$ is a feasible control path of the program in Fig. 3.6 as, under the standard interpretation of the arithmetic symbols, the sequence of states $\langle (0,0,0,0,0,0), (0,0,0,0,2,2), (1,100,1,10,1,2), (2,200,2,20,0,2)\rangle$ satisfies (3.4).

A location ℓ is *reachable* iff there exists a feasible control path ending with ℓ. P is *safe* iff ℓ^e is *not* reachable. For example, the program in Fig. 3.6 is safe, as shown in the previous section. P is said to be *decidable* iff the safety problem of P is decidable. For example, the program U obtained from P in Fig. 3.6 by replacing b with 5 is decidable because (a) U has finitely many feasible control paths, each of finite length, and (b) the formulas only use linear arithmetic (integers or rationals) which is decidable.

[6]This suffices as we only deal with safety properties.

We will now define invariance and proof of safety.

Definition 4 (Invariant Map)**.** *An* invariant map *for P is a map $\mathcal{I}$ from locations to sets of sentences over the signature $\mathcal{S} \cup V$ such that the following hold:*

1. $\models_{Th} \top \implies \bigwedge \pi(\ell^o)$, *and*
2. *for every* $\ell_i, \ell_j \in L$, $\models_{Th} (\bigwedge \pi(\ell_i) \ \wedge \ \tau(\ell_i, \ell_j)) \implies \bigwedge \pi(\ell_j)'$.

One can show, using a least fixed-point characterization, that given an invariant map $\mathcal{I}$, $\mathcal{I}(\ell)$ over-approximates the reachable states at a location ℓ.

Definition 5 (Safety Proof)**.** *A* safety proof *for P is an invariant map π that is also safe, i.e.,* $\models_{Th} \bigwedge \pi(\ell^e) \implies \bot$.

For example, Fig. 3.5(c) shows a safety proof for the program in Fig. 3.6, as also discussed in the previous section. As the formulas given by an invariant map over-approximate the reachable states, safety of the map implies safety of the program.

A *counterexample to safety* is a triple $\langle \bar{\ell}, I, \bar{s} \rangle$ where $\bar{\ell}$ is a feasible control path in P ending with ℓ^e, I is an $\mathcal{S}$-structure with $I \models Th$, and $\bar{s}$ is a sequence of states that satisfy (3.4). For example, $\langle \langle en, lp, lp, lp, lp, lp, er \rangle, I, \mathscr{C}_4 \rangle$, where I is the $\mathcal{S}$-structure that interprets the arithmetic symbols in the standard way and $\mathscr{C}_4$ is as shown in (3.3) is a counterexample to safety for the program $\hat{P}_1$ in Fig. 3.4(b).

Definition 6 (Abstraction Relation)**.** *Given two programs, $P_1 = \langle L_1, \ell_1^o, \ell_1^e, V_1, \tau_1 \rangle$ and $P_2 = \langle L_2, \ell_2^o, \ell_2^e, V_2, \tau_2 \rangle$, P_2 is an* abstraction *(i.e., an over-approximation) of P_1 via a total function $\sigma : L_1 \to L_2$, denoted $P_1 \preceq_\sigma P_2$, iff*

1. $V_1 = V_2$,
2. $\sigma(\ell_1^o) = \ell_2^o$ *and* $\sigma(\ell_1^e) = \ell_2^e$, *and*
3. *for every* $\ell_i, \ell_j \in L_1$, $\models_{Th} \tau_1(\ell_i, \ell_j) \implies \tau_2(\sigma(\ell_i), \sigma(\ell_j))$.

In this case, we also say that P_1 *is an* under-approximation (*i.e., a refinement*) *of* P_2 *and call* σ *an* abstraction function. *We say that* P_2 *strictly abstracts* P_1 *via* σ, *denoted* $P_1 \prec_\sigma P_2$, *iff* $P_1 \preceq_\sigma P_2$ *and there is no function* $\nu : L_2 \to L_1$ *such that* $P_2 \preceq_\nu P_1$. *When* σ *is clear from the context or unnecessary, we drop the subscript.*

That is, P_2 abstracts P_1 iff there is a total map σ from L_1 to L_2 such that every feasible transition of P_1 corresponds (via σ) to a feasible transition of P_2. For example, if P_1 is a finite unrolling of P_2, then σ maps the locations of P_1 to the corresponding ones in P_2. For an example of strict abstraction, it can be shown that $P \prec_{id} \hat{P}_1$, where P is in Fig. 3.6, $\hat{P}_1$ is in Fig. 3.4(a), and *id* denotes the identity relation.

One can easily show that the above notion of abstraction is proof-preserving, i.e., if $P_1 \preceq P_2$, then a safety proof π of P_2 is also a safety proof of P_1.

For two transition relations τ_1 and τ_2 over a set of locations L, we write $\tau_1 \implies \tau_2$ to denote that for every $\ell_1, \ell_2 \in L$, $\models_{Th} \tau_1(\ell_1, \ell_2) \implies \tau_2(\ell_1, \ell_2)$. We also write $\tau_1 \wedge \tau_2$ to denote the point-wise conjunction of the two transition relations.

We extend $\sigma : L_1 \to L_2$ from locations to control paths in the straightforward manner. For a counterexample $\mathscr{C} = \langle \bar{\ell}, I, \bar{s} \rangle$, we define $\sigma(\mathscr{C}) \equiv \langle \sigma(\bar{\ell}), I, \bar{s} \rangle$. For a transition relation τ on L_2, we write $\sigma(\tau)$ to denote an embedding of τ into L_1 via σ, defined as follows: for locations $\ell_1, \ell_2 \in L_1$, $\sigma(\tau)(\ell_1, \ell_2) = \tau(\sigma(\ell_1), \sigma(\ell_2))$. For example, in the definition above, if $P_1 \preceq_\sigma P_2$, then $\tau_1 \implies \sigma(\tau_2)$.

In the following, to avoid clutter, we assume a fixed $\mathcal{S}$-theory Th and we write $\models$ to mean $\models_{Th}$. Also, every $(\mathcal{S} \cup X)$-structure we consider, for an arbitrary set of new symbols X, is assumed to be a model of Th.

3.4 The Algorithm

In this section, we describe our algorithm SPACER at a high-level, where the description of several routines is confined only to their interfaces. Our specific implementation choices are described in Section 3.5. Figs. 3.7 and 3.8 show the pseudo-code of the algorithm. The routine SPACER checks safety of a given program $P = \langle L_P, \ell^o_P, \ell^e_P, V_P, \tau_P \rangle$. The algorithm maintains (a) an invariant map $\mathcal{I}$ (see Definition 4), (b) an abstraction A of P, (c) a decidable under-approximation U of A, and (d) a function σ such that $U \preceq_\sigma A$. The abstraction A has the same set of control locations as P and satisfies $P \preceq_{id} A$, i.e., A differs from P only in its transition relation. We write $A_\mathcal{I}$ to denote the restriction of A to the invariant map $\mathcal{I}$, obtained by strengthening τ_A to $\lambda \ell_1, \ell_2 \cdot \mathcal{I}(\ell_1) \wedge \tau_A(\ell_1, \ell_2) \wedge \mathcal{I}(\ell_2)'$. Similarly, we write $U_{\mathcal{I},A}$ to denote the restriction of U to the invariant map $\mathcal{I}$ of A, obtained by strengthening τ_U to $\lambda \ell_1, \ell_2 \cdot \mathcal{I}(\sigma(\ell_1)) \wedge \tau_U(\ell_1, \ell_2) \wedge \mathcal{I}(\sigma(\ell_2))'$. When A is clear, we simply write $U_\mathcal{I}$. SPACER assumes the existence of an oracle, SOLVE, that decides whether $U_\mathcal{I}$ is safe and returns either a safety proof or a counterexample (see Section 3.5 for an implementation of SOLVE).

SPACER initializes the abstraction A of P and an under-approximation U of A, using INITABS and INITUNDER, respectively (lines 1–2). It then initializes the invariant map $\mathcal{I}$ to the empty map (line 3). Each iteration of the main loop (at line 4) checks whether $U_\mathcal{I}$ is safe, for the current values of U and $\mathcal{I}$, using SOLVE (line 5). If $U_\mathcal{I}$ is safe with a proof π, it is then checked whether a safety proof of the original program P can be obtained using π, as follows. First, π is mined for new invariants of P using EXTRACTINVS (line 7). Then, if the invariants at the

error location ℓ_P^e are unsatisfiable, it means that the error location is shown to be unreachable and SPACER returns SAFE (lines 8–9). Otherwise, the abstraction A is updated to a new *Proof Based Abstraction* via PBA, and a new under-approximation is constructed using NEXTUNDER (lines 10–11). If, on the other hand, $U_{\mathcal{I}}$ is unsafe at line 4, the obtained counterexample $\mathscr{C}$ is validated using CEGAR (line 13). If $\mathscr{C}$ is feasible in P, SPACER returns UNSAFE (line 15); otherwise, both A and U are refined (see the description of CEGAR below).

In the following, we give a brief description of the routines. Let $U = \langle L_U, \ell_U^o, \ell_U^e, V_U, \tau_U \rangle$, $U \preceq_\sigma A$, and $A = \langle L_A, \ell_A^o, \ell_A^e, V_A, \tau_A \rangle$. Note that $L_A = L_P$ as mentioned above. **EXTRACTINVS** has two high level steps: (a) use the proof π of U to obtain a conjunction of formulas at each location of P, and (b) compute the maximal subset of those conjuncts at each location that are together invariant (according to Definition 4). To obtain the conjunctions at a given location in L_P, we collect the formulas given by the proof π of U at all corresponding locations (w.r.t. σ) in L_U (lines 17–18), take their disjunction, and convert to conjunctive form (not necessarily conjunctive *normal* form; see lines 19–20). The intuition behind taking a disjunction is that the various locations $\ell_u \in L_U$ with $\sigma(\ell_u) = \ell \in L_P$ represent different subsets (not necessarily exhaustive) of the reachable states at ℓ. In our implementation, we only have one such corresponding location (see Section 3.5). To obtain the maximal subsets that are invariant, we use a straightforward greatest fixed-point computation (lines 21–22), similar to the HOUDINI approach [54].

```
global(program P)
global(invariant map 𝓘 of P)

Spacer()
1   A ← InitAbs(P)
2   (U, σ) ← InitUnder(A)
3   𝓘 ← empty map
4   while true do
5       (result, π, 𝒞) ← Solve(U_𝓘)
6       if result is Safe then
7           𝓘 ← 𝓘 ∪ ExtractInvs(A, U, σ, π)
8           if ⋀𝓘(ℓ^e_P) is unsatisfiable then
9               return Safe
10          (A, U) ← Pba(A, U, σ, π)
11          (U, σ) ← NextUnder(A, U, σ)
12      else
13          (feas, A, U) ← Cegar(A, U, σ, 𝒞)
14          if feas then
15              return Unsafe

ExtractInvs(A, U, σ, π)
16  R, C ← empty maps from locations in L_P to sets of sentences over S ∪ V_P
17  for ℓ ∈ L_U do
18      add ⋀π(ℓ) to C(σ(ℓ))
19  for ℓ ∈ L_P do
20      R(ℓ) ← conjuncts(⋁C(ℓ))
21  while exist ℓ_i, ℓ_j ∈ L_P, φ ∈ R(ℓ_j) s.t. ⊭ (R(ℓ_i) ∧ 𝓘(ℓ_i) ∧ τ_P(ℓ_i, ℓ_j) ⇒ φ(V'_P)) do
22      R(ℓ_j) := R(ℓ_j) \ {φ}
23  return R

NextUnder(A, U, σ)
24  return Û s.t. U ≺_{σ_1} Û ⪯_{σ_2} A, σ = σ_2 ∘ σ_1 and
      Strengthen(U, σ(τ_P)) ≺ Strengthen(Û, σ_2(τ_P))
```

Figure 3.7: Pseudo-code of Spacer, except the routines for Pba and Cegar which are given in Fig. 3.8.

PBA updates the abstraction A of P by using the safety proof π of U as follows. As $U \preceq_\sigma A$, $\tau_U \implies \sigma(\tau_A)$ holds and assume without loss of generality that τ_U is of the form $\sigma(\tau_A) \wedge \rho$, for some ρ (i.e., one can always equivalently rewrite τ_U to this form). Let W be obtained from U by strengthening the transition relation with $\sigma(\tau_P)$ (using STRENGTHEN on line 26). Clearly, $\tau_W \equiv \sigma(\tau_P) \wedge \rho$. It is easy to see that π is also a proof of W. An abstraction $\hat{W}$ of W is then obtained such that (a) $\tau_{\hat{W}} = \sigma(\hat{\tau}_P) \wedge \hat{\rho}$ where $\hat{\tau}_P$ and $\hat{\rho}$ respectively abstract τ_P and ρ, (b) for the new abstraction $\hat{A}$ obtained by replacing the transition relation of A with $\hat{\tau}_P$, π is a safety proof of $\hat{W}_{\mathcal{I},\hat{A}}$ (line 27). That is, $\hat{A}$ is obtained as a proof-based abstraction of P using the proof π of U and the currently known invariants $\mathcal{I}$.

NEXTUNDER returns the next under-approximation $\hat{U}$ of A to be checked for safety. We require that the abstraction functions between U, $\hat{U}$, and A compose so that the corresponding transitions in U and $\hat{U}$ map to the same transition of the common abstraction A. To ensure progress, we require $U \prec \hat{U}$. Moreover, to ensure progress in checking safety of P, we also require the last condition on line 24. Intuitively, we require $\hat{U}$ to also have more *concrete behaviors* than U. If this were not possible, safety of U would have implied safety of P and SPACER would have terminated.

CEGAR checks if the counterexample $\mathscr{C}$ exhibits a feasible behavior in P, using ISFEASIBLE (line 29). If $\mathscr{C}$ is feasible, CEGAR returns saying so (line 34). Otherwise, $\mathscr{C}$ is spurious and the abstraction $\hat{A}$ is refined to A by eliminating $\mathscr{C}$ (and possibly more spurious behaviors) (line 31). This is obtained by strengthening the transition relation, i.e., $A \prec_{id} \hat{A}$ holds. Finally, the under-approximation $\hat{U}$ is strengthened with the refined transition relation of A (using STRENGTHEN on line 32), such that

```
    PBA(A, U, σ, π)
25  |  let ρ be s.t. τ_U ≡ σ(τ_A) ∧ ρ
26  |  W ← STRENGTHEN(U, σ(τ_P))
27  |  choose Ŵ s.t. W ⪯ Ŵ and τ_Ŵ = σ(τ̂_P) ∧ ρ̂ with τ_P ⟹ τ̂_P, ρ ⟹ ρ̂, π is a safety
    |  proof of Ŵ_{I,Â}, where Â = A[τ_A ← τ̂_P]
28  |  return (Â, Ŵ)

    CEGAR(Â, Û, σ, 𝒞)
29  |  feas ← ISFEASIBLE(σ(𝒞), P)
30  |  if not feas then
31  |  |  let A ≺_id Â s.t. ¬ISFEASIBLE(σ(𝒞), A_I)
32  |  |  U ← STRENGTHEN(Û, σ(τ_A))
33  |  |  return (false, A, U)
34  |  return (true, None, None)
```

Figure 3.8: Routines for Proof based Abstraction and CEGAR.

the resulting U satisfies $U \preceq_\sigma A$ for the same abstraction function σ.

In the following, we show the soundness and progress guarantees of SPACER.

Lemma 7 (Invariant Maps)**.** *$\mathcal{I}$ is always an invariant map for P.*

Proof. Initially, $\mathcal{I}$ is empty and $\mathcal{I}(\ell) = \top$ for every location $\ell \in L_P$. Clearly, $\mathcal{I}$ is an invariant map. When EXTRACTINVS returns on line 23, the guard of the while loop at line 21 fails to hold which implies that the map $\mathcal{R}$ is also an invariant map. It follows that when $\mathcal{I}$ is updated on line 7, it remains an invariant map. □

Soundness is now immediate:

Theorem 8 (Soundness)**.** *P is safe (unsafe) if* SPACER *returns* SAFE (UNSAFE)*.*

To show progress, we start with a useful lemma. In the following, we sometimes refer to the components of a program P by application, in addition to using subscripts, e.g., $L(P)$ denotes the locations of P.

Lemma 8. *Let $U_1 \preceq_{\sigma_1} A$ and let ρ_1 be such that $\tau(U_1) \equiv \sigma_1(\tau_A) \wedge \rho_1$. If $U_1 \preceq_{\mu} U_2 \preceq_{\sigma_2} A$ with $\sigma_1 = \sigma_2 \circ \mu$, then there exists ρ_2 such that $\tau(U_2) \equiv \sigma_2(\tau_A) \wedge \rho_2$ and $\rho_1 \implies \mu(\rho_2)$.*

Proof. As $U_2 \preceq_{\sigma_2} A$, we know that $\tau(U_2) \implies \sigma_2(\tau_A)$ (Definition 6). So, there exists ρ such that $\tau(U_2) \equiv \sigma_2(\tau_A) \wedge \rho$. As $U_1 \preceq_{\mu} U_2$, we also know that $\tau(U_1) \implies \mu(\tau(U_2))$. Together with $\sigma_1 = \sigma_2 \circ \mu$, we obtain

$$\sigma_1(\tau_A) \wedge \rho_1 \implies \sigma_1(\tau_A) \wedge \mu(\rho). \tag{3.5}$$

Consider

$$\rho_2 = \rho \vee \lambda \ell_i^2, \ell_j^2 \in L(U_2) \cdot \left(\bigvee_{\ell_i^1, \ell_j^1 \in L(U_1)} \mu(\ell_i^1) = \ell_i^2 \wedge \mu(\ell_j^1) = \ell_j^2 \wedge \rho_1(\ell_i^1, \ell_j^1) \right).$$

Intuitively, ρ_2 captures all the transitions that must be feasible in U_2 as guaranteed by the relationship $U_1 \preceq_{\mu} U_2$.

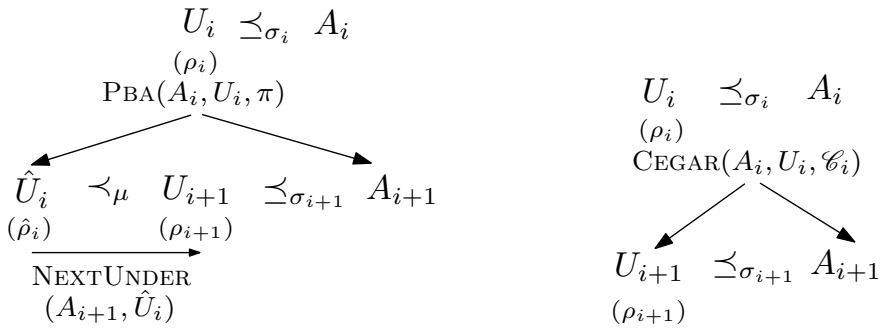
It can be easily shown that $\rho_1 \implies \mu(\rho_2)$.

It remains to show that $\sigma_2(\tau_A) \wedge \rho_2$ is equivalent to $\tau(U_2) \equiv \sigma_2(\tau_A) \wedge \rho$. That is, we need to show that, for all $\ell_i^2, \ell_j^2 \in L(U_2)$,

$$\models (\sigma_2(\tau_A) \wedge \rho)\,(\ell_i^2, \ell_j^2) \iff (\sigma_2(\tau_A) \wedge \rho_2)\,(\ell_i^2, \ell_j^2).$$

The left to right direction is obvious as $\models \rho \implies \rho_2$ from the definition of ρ_2.

For the other direction, assume for the sake of contradiction that there exist $\ell_i^2, \ell_j^2 \in L(U_2)$ and an $\mathcal{S} \cup V \cup V'$-structure I (with $I \models Th$) such that $I \models$



(a) U_i is safe. (b) U_i is unsafe.

Figure 3.9: Relation between two successive under-approximations U_i and U_{i+1}.

$(\sigma_2(\tau_A) \wedge \rho_2)\,(\ell_i^2, \ell_j^2)$ and $I \not\models \rho(\ell_i^2, \ell_j^2)$. From the definition of ρ_2, it follows that there exist $\ell_i^1, \ell_j^1 \in L(U_1)$ such that $I \models \rho_1(\ell_i^1, \ell_j^1)$, and $\mu(\ell_i^1) = \ell_i^2$ and $\mu(\ell_j^1) = \ell_j^2$. Moreover, we know that $I \models \tau_A(\sigma_2(\ell_i^2), \sigma_2(\ell_j^2))$. As, $\sigma_1 = \sigma_2 \circ \mu$, it follows that $I \models \tau_A(\sigma_1(\ell_i^1), \sigma_1(\ell_j^1))$. So, we have found $\ell_i^1, \ell_j^1 \in L(U_1)$ such that $I \models (\sigma_1(\tau_A) \wedge \rho_1)\,(\ell_i^1, \ell_j^1)$. From (3.5), it follows that $I \models \mu(\rho)(\ell_i^1, \ell_j^1)$, i.e., $I \models \rho(\ell_i^2, \ell_j^2)$ which contradicts our assumption. □

Theorem 9 (Progress)**.** *Let A_i, U_i, and $\mathscr{C}_i$ be the values of A, U, and $\mathscr{C}$ in the ith iteration of* SPACER *with $U_i \preceq_{\sigma_i} A_i$ and let $\dot{U}_i$ denote the concretization of U_i, i.e., $\dot{U}_i =$ STRENGTHEN$(U_i, \sigma_i(\tau_P))$. Then, if U_{i+1} exists,*

1. *if U_i is safe, then U_{i+1} has strictly more concrete behaviors, i.e., $\dot{U}_i \prec \dot{U}_{i+1}$,*
2. *if U_i is unsafe, U_{i+1} has the same concrete behaviors, i.e., $\dot{U}_i \preceq_{id} \dot{U}_{i+1}$ and $\dot{U}_{i+1} \preceq_{id} \dot{U}_i$, and*
3. *if U_i is unsafe, $\mathscr{C}_i$ does not repeat in future, i.e., for every $j > i, \sigma_j(\mathscr{C}_j) \neq \sigma_i(\mathscr{C}_i)$.*

Proof. 1. U_{i+1} is obtained from U_i after a call to PBA followed by NEXTUNDER, as shown in Fig. 3.9(a). For U_j, the figure also shows ρ_j in brackets such that

$\tau(U_j) \equiv \sigma_j(\tau(A_j)) \wedge \rho_j$ (this is always possible as $\tau(U_j) \implies \sigma_j(\tau(A_j))$). PBA ensures that $\rho_i \implies \hat{\rho}_i$ and Lemma 8 guarantees the existence of a ρ_{i+1} with $\hat{\rho}_i \implies \mu(\rho_{i+1})$. Together, we have $\rho_i \implies \mu(\rho_{i+1})$. Furthermore, NEXTUNDER requires $\sigma_i = \sigma_{i+1} \circ \mu$. Then, $\dot{U}_i \preceq_\mu \dot{U}_{i+1}$, as shown below.

$$\begin{aligned}
\tau(\dot{U}_i) &\equiv \sigma_i(\tau_P) \wedge \rho_i \\
&\implies (\sigma_{i+1} \circ \mu)(\tau_P) \wedge \mu(\rho_{i+1}) \\
&\implies \mu(\sigma_{i+1}(\tau_P)) \wedge \mu(\rho_{i+1}) \\
&\implies \mu(\tau(\dot{U}_{i+1}))
\end{aligned}$$

To show that $\dot{U}_i \prec \dot{U}_{i+1}$, assume for the sake of contradiction that $\dot{U}_{i+1} \preceq_\omega \dot{U}_i$. Then, as $\rho_i \implies \hat{\rho}_i$,

$$\begin{aligned}
\tau(\dot{U}_{i+1}) &\equiv \sigma_{i+1}(\tau_P) \wedge \rho_{i+1} \\
&\implies \omega(\sigma_i(\tau_P) \wedge \rho_i) \\
&\implies \omega(\sigma_i(\tau_P) \wedge \hat{\rho}_i) \\
&\equiv \omega(\tau(\dot{\hat{U}}_i))
\end{aligned}$$

giving us $\dot{U}_{i+1} \preceq_\omega \dot{\hat{U}}_i$. This contradicts $\dot{\hat{U}}_i \prec \dot{U}_{i+1}$ on line 24 of Fig. 3.7.

2. U_{i+1} is obtained from U_i after a call to CEGAR as shown in Fig. 3.9(b). Again, for U_j, the figure shows ρ_j in brackets such that $\tau(U_j) \equiv \sigma_j(\tau(A_j)) \wedge \rho_j$. CEGAR

ensures that $\rho_i = \rho_{i+1}$ and $\sigma_i = \sigma_{i+1}$. These imply that $\tau(\dot{U}_i) \iff \tau(\dot{U}_{i+1})$.

3. Let $\mathscr{C}_i = \langle \bar{\ell}_i, I, \bar{s} \rangle$. We prove the stronger statement that for every $j > i$, there exist a control path $\bar{\ell}_j$ in U_j and a ρ_j such that $\tau(U_j) \equiv \sigma_j(\tau(A_j)) \wedge \rho_j$, and the following hold:

   (a) $\sigma_j(\bar{\ell}_j) = \sigma_i(\bar{\ell}_i)$,

   (b) $\langle \bar{\ell}_j, I, \bar{s} \rangle$ is a counterexample for U_j when the transition relation is restricted to ρ_j but not when restricted to $\sigma_j(\tau(A_j))$.

   In words, we show that the control path of $\mathscr{C}_i$ corresponds to a control path in every future U_j (via σ_j) and the state sequence $\bar{s}$ is feasible when restricted to ρ_j but not when restricted to $\sigma_j(\tau(A_j))$. The latter is sufficient to show that U_j does not admit $\mathscr{C}_i$.

   We prove this by induction on j. If $j = i + 1$, Fig. 3.9(b) shows the relation between U_i and U_{i+1}. Again, CEGAR ensures that $\rho_i = \rho_{i+1}$ and $\sigma_i = \sigma_{i+1}$. The required control path $\bar{\ell}_j$ in (a) is the same as $\bar{\ell}_i$. Also, CEGAR ensures that $\tau(A_j)$ does not admit $\mathscr{C}_i$, satisfying (b).

   Now, assume that U_j satisfies (a) and (b), for an arbitrary j. We show that U_{j+1} also satisfies (a) and (b). If U_{j+1} is obtained from U_j after a call to CEGAR, the argument is the same as for the base case above. The other possibility is as shown in Fig. 3.9(a) where U_j is safe and U_{j+1} is obtained after a call to PBA, followed by a call to NEXTUNDER. Consider ρ_j, $\hat{\rho}_j$ and ρ_{j+1}, similar to ρ_i, $\hat{\rho}_i$ and ρ_{i+1} in the figure. Note that PBA ensures that $\rho_j \implies \hat{\rho}_j$.

   To see that (a) is satisfied, consider the control path $\mu(\bar{\ell}_j)$ and note that $\sigma_j = \sigma_{j+1} \circ \mu$ (line 24 of Fig. 3.7).

To see that (b) is satisfied, Lemma 8 ensures the existence of ρ_{j+1} with $\hat{\rho}_j \implies \mu(\rho_{j+1})$. As $\bar{s}$ is feasible along $\bar{\ell}_j$ when the transition relation is restricted to ρ_j and hence, for $\hat{\rho}_j$, it is also feasible along $\mu(\bar{\ell}_j)$ when the transition relation is restricted to $\mu(\rho_{j+1})$. Moreover, $\bar{s}$ is infeasible along $\bar{\ell}_j$ for the restriction $\sigma_j(\tau(A_{j+1}))$, as $\hat{U}_j$ is safe. Hence, it remains infeasible along $\mu(\bar{\ell}_j)$ for the restriction $\sigma_{j+1}(\tau(A_{j+1}))$ (follows from $\sigma_{j+1} \circ \mu = \sigma_j$).

□

In this section, we presented the high-level structure of SPACER. As we have seen above, we only presented an interface for the routines INITUNDER, EXTRACTINVS, PBA, NEXTUNDER, CEGAR, ISFEASIBLE. In the next section, we complete the picture by describing the implementation used in our prototype.

3.5 Implementation

Let $P = \langle L, \ell^o, \ell^e, V, \tau \rangle$ be the input program. First, we transform P to $\tilde{P}$ by creating new *counter* variables for the loops of P and adding extra constraints to the transition relation in order to count the number of loop iterations, as described below.

As the first step, we construct a *Weak Topological Order* (WTO) [24] of P, which is a well-parenthesized total order of its locations L without two consecutive open brackets, denoted $<$, satisfying the following condition. Let the locations within a matching open-close bracket pair constitute a *component* and let the smallest location w.r.t $<$ in a component be its *head*. Let $hds(\ell)$ be the outside-in list of the heads of

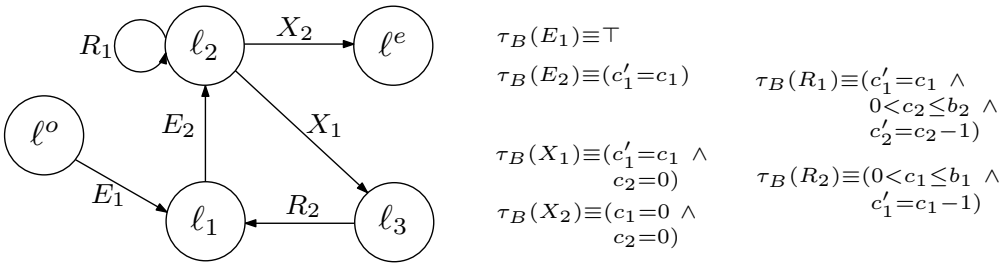


Figure 3.10: Program with a nested loop and its corresponding *bounded* transition constraints.

components containing ℓ. Let $\ell_1 \leq \ell_2$ be defined as $(\ell_1 = \ell_2 \vee \ell_1 < \ell_2)$. Then,

$$\forall \ell_i, \ell_j \in L \cdot \tau(\ell_i, \ell_j) \wedge \ell_j \leq \ell_i \implies \ell_j \in \mathit{hds}(\ell_i) \tag{3.6}$$

Intuitively, $<$ is a total order of L such that each component identifies a loop in P, the head of a component identifies the entry location of the loop and $\mathit{hds}(\ell)$ identifies the outside-in list of nested loops containing ℓ. Condition (3.6) says that a *back-edge*, w.r.t $<$, leads to the head of a component containing the source of the edge, denoting the start of a new iteration of the corresponding loop. For example, Fig. 3.10 shows a program with two loops, an outer loop $\langle \ell_1, \ell_2, \ell_3 \rangle$ and an inner loop $\langle \ell_2 \rangle$. One possible WTO for this program is "$\ell^o(\ell_1(\ell_2)\ell_3)\ell^e$" with ℓ_1 and ℓ_2 as the heads of the two components.

Note that the above definition of WTO is non-deterministic and there are multiple ways of implementing such an ordering. Without loss of generality, assume that ℓ^o is always the smallest and ℓ^e is always the largest location of a WTO.

Bound Variables. Next, we introduce a set C of rational variables, one per head of a component, and the corresponding partial mapping $\mathit{ctr} : L \rightharpoonup C$. Intuitively, $\mathit{ctr}(\ell)$ is the number of iterations (completed or remaining, depending on whether we are

counting up or down, respectively) of the component whose head is ℓ. Also, let B be another set of rational variables and let $bound : C \to B$ be a bijection (i.e., $|B| = |C|$). Informally, $bound(c)$ denotes the upper bound of c. For example, in Fig. 3.10 we have $C = \{c_1, c_2\}$, $c_1 = ctr(\ell_1)$, $c_2 = ctr(\ell_2)$, $B = \{b_1, b_2\}$, $bound(c_1) = b_1$, and $bound(c_2) = b_2$. We construct a *bounded* program $\tilde{P} = \langle L, \ell^o, \ell^e, V \cup C \cup B, \tilde{\tau} \rangle$, where $\forall \ell_i, \ell_j \in L \cdot \tilde{\tau}(\ell_i, \ell_j) = \tau(\ell_i, \ell_j) \wedge \tau_B(\ell_i, \ell_j)$ and $\tau_B(\ell_i, \ell_j)$ is a set of constraints defined as follows. Let $c_j = ctr(\ell_j)$ and $b_j = bound(c_j)$. We only describe the constraints for counting down the number of iterations from an initial value, respecting the bounded given by the bounding variables.

Entry: $\ell_i < \ell_j$ and ℓ_j is a head, i.e., entering a new component (e.g., E_1 and E_2 in Fig. 3.10). Then, $\tau_B(\ell_i, \ell_j)$ contains a constraint corresponding to c_j being assigned non-deterministically.

Re-entry: $\ell_j \leq \ell_i$, i.e., re-entering a component via a back-edge (e.g., R_1 and R_2 in Fig. 3.10). Then, $\tau_B(\ell_i, \ell_j)$ contains the constraint $(0 \leq c'_j \wedge c'_j = c_j - 1 \wedge c_j \leq b_j$, i.e., it decrements c_j as long as it is not zero.

Exit: $\ell_i < \ell_j \wedge hds(\ell_i) \supset hds(\ell_j)$, i.e., exiting (one or more) components containing ℓ_i (e.g., X_1 and X_2 in Fig. 3.10). Then, for each $h \in hds(\ell_i) \setminus hds(\ell_j)$, $\tau_B(\ell_i, \ell_j)$ contains the constraint $ctr(h) = 0$.

Pass-on. For each $h \in hds(\ell_j) \setminus \{\ell_j\}$, $\tau_B(\ell_i, \ell_j)$ contains the constraint $ctr(h) = ctr(h)'$. Thus, when the transition is inside a component the current value of its counter is remembered. See τ_B for the transitions E_2, R_1 and X_1 in Fig. 3.10.

In other words, a counter is assigned a non-deterministic initial value when entering its component, and decremented until zero before exiting. Since the bound

variables (i.e., B) are unconstrained, $\tilde{P}$ and P are equivalent w.r.t. safety, as shown below.

Lemma 9. *P is safe iff $\tilde{P}$ is safe.*

Proof. To show that safety of P implies safety of $\tilde{P}$, we prove its contrapositive. Assume $\tilde{P}$ is unsafe. So, there exists a counterexample to safety $\langle \bar{\ell}, I, \bar{s} \rangle$. As $\tau_{\tilde{P}} = \tau \wedge \tau_B$, it is obvious that projecting the state sequence onto V gives us a counterexample to safety for P. So, P is unsafe as well.

Now suppose that P is unsafe with a counterexample $\mathscr{C} = \langle \bar{\ell}, I, \bar{s} \rangle$. Then, we can extend $\mathscr{C}$ to a counterexample $\tilde{\mathscr{C}}$ of $\tilde{P}$ as follows: (a) for each loop of P and for each time the loop is entered and exited along $\mathscr{C}$, count the number of iterations, say n, (b) assign values to the corresponding counter variables along the control path to simulate a count-down from n to 0, and (c) for the bound variable corresponding to the loop, assign a value greater than or equal to the maximum number of iterations of the loop along $\mathscr{C}$. So, $\tilde{\mathscr{C}}$ is also unsafe. □

Now, given a safety proof of $\tilde{P}$, one can transform it to a proof of P by universally quantifying all the bound and counter variables. See Appendix 3.A for a proof. We can thus check safety of $\tilde{P}$ in order to decide safety of P.

In the rest of this section, we describe our abstractions and under-approximations of $\tilde{P}$, followed by our implementation of the various routines in Fig. 3.7.

Abstractions. Recall that $\tau_{\tilde{P}} = \tau \wedge \tau_B$. Let Σ be a set of fresh Boolean variables not appearing in τ or τ_B. For every pair of locations $\ell_1, \ell_2 \in L$, we will now transform $\tau(\ell_1, \ell_2)$ to the equivalent $\exists \Sigma \cdot (\tau_\Sigma(\ell_1, \ell_2) \wedge \bigwedge \Sigma)$ such that the variables in Σ

only appear negatively in τ_Σ. We refer to Σ as *assumptions* following SAT terminology [47]. Note that dropping some assumptions from the conjunction $\bigwedge \Sigma$ results in an abstract transition relation, i.e., $\models \tau(\ell_1, \ell_2) \implies \exists \Sigma \cdot \left(\tau_\Sigma(\ell_1, \ell_2) \wedge \bigwedge \hat{\Sigma}\right)$ for $\hat{\Sigma} \subseteq \Sigma$. We write $\hat{\tau}(\hat{\Sigma})$ to denote the resulting abstract transition relation. So, we have $\tau \implies \hat{\tau}(\hat{\Sigma})$.

Note that $\hat{\tau}(\hat{\Sigma})$ can be obtained from τ_Σ by substituting the assumptions in $\hat{\Sigma}$ by $\top$ and the rest of the assumptions by $\bot$, i.e., $\hat{\tau}(\hat{\Sigma}) = \tau_\Sigma[\hat{\Sigma} \leftarrow \top, \Sigma \backslash \hat{\Sigma} \leftarrow \bot]$. The only abstractions of $\tilde{P}$ we consider are the ones which abstract τ and keep τ_B unchanged. That is, every abstraction $\hat{P}$ of $\tilde{P}$ is such that $\tilde{P} \preceq_{id} \hat{P}$ with $\tau_{\hat{P}} = \hat{\tau}(\hat{\Sigma}) \wedge \tau_B$ for some $\hat{\Sigma} \subseteq \Sigma$. Given $\hat{\Sigma} \subseteq \Sigma$, we denote the corresponding abstraction as $\tilde{P}(\hat{\Sigma})$.

Under-approximations. Given an abstraction $\tilde{P}(\hat{\Sigma})$ for a subset of assumptions $\hat{\Sigma} \subseteq \Sigma$, an under-approximation is obtained by constraining the bound variables in B. In particular, an under-approximation $U(\hat{\Sigma}, \mathit{bvals})$ for a total map $\mathit{bvals} : B \to \mathbb{N}$ from B to natural numbers is obtained by strengthening τ_B with the constraints $\bigwedge_{b \in B} b \leq \mathit{bvals}(b)$, for every pair of locations. We denote the strengthening of τ_B by $\tau_B(\mathit{bvals})$. So, the under-approximation $U(\hat{\Sigma}, \mathit{bvals})$ satisfies $U(\hat{\Sigma}, \mathit{bvals}) \preceq_{id} \tilde{P}(\hat{\Sigma})$, with the transition relation $\hat{\tau}(\hat{\Sigma}) \wedge \tau_B(\mathit{bvals})$.

Solve. To implement Solve (see Fig. 3.7) we first transform $U_\mathcal{I}$, the restriction of the current under-approximation U to the invariants $\mathcal{I}$, to Horn-SMT [74], which essentially encodes the verification conditions of the program by using predicate variables to denote the unknown invariants. The Horn-SMT problem is then passed to the tool Z3 [45]. While Z3 is primarily an SMT solver, it also has the capability of solving Horn-SMT [74]. Note that, in presence of an oracle for the $\mathcal{S}$-theory *Th*

Global	*Trans*	$E_{i,j} \implies \tau_\Sigma(\ell_i, \ell_j) \wedge \tau_B(\ell_i, \ell_j), \quad \ell_i, \ell_j \in L$	(1)
		$N_i \implies \bigvee_j E_{j,i}, \quad \ell_i \in L$	(2)
	Invars	$\left(\bigvee_j E_{i,j}\right) \implies \varphi, \quad \ell_i \in L, \varphi \in \mathcal{I}(\ell_i)$	(3)
		$N_i \implies \varphi', \quad \ell_i \in L, \varphi \in \mathcal{I}(\ell_i)$	(4)
Local	*Lemmas*	$\bigwedge_{\ell_i \in L, \varphi \in \pi(\ell_i)} \left(\mathcal{A}_{\ell_i,\varphi} \implies \left(\left(\bigvee_j E_{i,j}\right) \implies \varphi\right)\right)$	(5)
		$\neg \bigwedge_{\ell_i \in L, \varphi \in \pi(\ell_i)} (\mathcal{B}_{\ell_i,\varphi} \implies (N_i \implies \varphi'))$	(6)
	Assump. Lits	$\mathcal{A}_{\ell,\varphi}, \quad \ell \in L, \varphi \in \pi(\ell)$	(7)
		$\neg\mathcal{B}_{\ell,\varphi}, \quad \ell \in L, \varphi \in \pi(\ell)$	(8)
	Concrete	Σ	(9)
	Bound Vals	$b \leq \mathit{bvals}(b), \quad b \in B$	(10)

Figure 3.11: Constraints used in our implementation of SPACER.

(e.g., linear arithmetic), U is decidable as the length of a feasible path is bounded. However, Z3 also has heuristics for solving undecidable problems. See Section 3.6 for a comparison between SPACER and Z3. Note that one can replace Z3 with any other tool that solves Horn-SMT problems (for e.g., our implementation of the algorithm described in Chapter 2).

Finally, we describe how to implement SPACER efficiently using an incremental SMT solver where constraints can be dynamically added or retracted for checking satisfiability of multiple instances. We implement the routines of SPACER in Fig. 3.7 by maintaining a set of constraints $\mathcal{C}$. At a high level, there are two types of constraints, as shown in Fig. 3.11. The *Global* constraints are global to all the routines and $\mathcal{C}$ is updated whenever a new global constraint is inferred by a routine. The *Local* constraints are local to a routine which are added to or retracted from $\mathcal{C}$ as needed. We will explain the various constraints below.

The global constraints labeled *Trans* encode the transition relation of $\tilde{P}$ using fresh Boolean variables $E_{i,j}$ and N_i for transitions and locations, respectively. The intended meaning of the Boolean variables $E_{i,j}$ and N_i is as follows: (a) setting $E_{i,j}$ to true *enables* the transition from the location ℓ_i to the location ℓ_j, also implying that the *current* location is ℓ_i, and (b) setting N_i to true means that the *next* location is ℓ_i. The constraints in (1) (see the figure) encode $\tau_{\tilde{P}}$ while leaving out the assumptions in Σ. So, choosing an abstraction of $\tilde{P}$ amounts to adding a subset of the Boolean variables in Σ as additional constraints. The constraints in (2) enforce that a location is reachable only via one of its incoming edges.

The global constraints labeled *Invars* encode the currently known invariants. In order to specify that the invariants hold before and after a transition, we encode the invariants in terms of both current-state variables (3) and next-state variables (4). To identify that the current location is ℓ_i, the antecedent in (3) specifies that at least one outgoing transition from ℓ_i is enabled. Similarly, to identify that the next location is ℓ_i, N_i is used as the antecedent in (4).

For a set of Boolean literals $\mathcal{A}$, let $\textsc{Sat}(\mathcal{C}, \mathcal{A})$ be a function that checks whether $\mathcal{C} \cup \mathcal{A}$ is satisfiable, and returns either a satisfying assignment or an *unsat core* $\hat{\mathcal{A}} \subseteq \mathcal{A}$ such that $\mathcal{C} \cup \hat{\mathcal{A}}$ is unsatisfiable. Modern SAT and SMT solvers, including Z3, support this functionality and the Boolean literals in $\mathcal{A}$ are called *assumption literals* [47].

The local constraints are explained along with the implementations of the various routines below.

InitAbs. is implemented by choosing $\tilde{P}$ as the initial abstraction, i.e., the initial subset $\hat{\Sigma}$ of Σ is Σ itself.

InitUnder. is implemented by first initializing *bvals* to $\lambda b \in B \cdot 0$ and then choosing $U(\Sigma, \mathit{bvals})$ as the initial under-approximation.

ExtractInvs is implemented by ExtractInvsImpl shown in Fig. 3.12. Let π be a safety proof of the current under-approximation. ExtractInvImpl extracts a *Maximal Inductive Subset* (MIS) of the formulas given by π w.r.t. the concrete transition relation $\tau \wedge \tau_B$ of $\tilde{P}$, which we explained intuitively in Section 3.2. To concretize the transition relation, we first add the constraints under *Concrete* in Fig. 3.11 to $\mathcal{C}$, i.e., we add all the assumptions in Σ. Then, we add the constraints under *Lemmas* to $\mathcal{C}$ which encode the formulas given by π over current and next-state variables guarded by fresh Boolean variables $\mathcal{A}_{\ell,\varphi}$ and $\mathcal{B}_{\ell,\varphi}$ for every location ℓ and formula $\varphi \in \pi(\ell)$. The negation in (6) is used to encode the negation of invariance (Definition 4) so that we can use SAT solving to check validity. We simulate the greatest fixed-point computation shown on lines 21–22 in Fig. 3.7 by iteratively enabling and disabling these Boolean variables as follows.

The MIS of π corresponds to the *maximal* subset $I \subseteq \{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi}$ such that $\mathcal{C} \cup I \cup \{\neg\mathcal{B}_{\ell,\varphi} \mid \mathcal{A}_{\ell,\varphi} \notin I\}$ is unsatisfiable. Intuitively, I selects a subset of the formulas from $\pi(\ell)$ over the current-state variables, for every location ℓ, that are together invariant (Definition 4). Now, in order to disable every other formula over the next-state variables, we also need to assert $\neg\mathcal{B}_{\ell,\varphi}$ where φ is a formula that is not invariant. I is computed by ExtractInvsImpl in Fig. 3.12. Each iteration of ExtractInvsImpl refines the set of formulas by eliminating the ones that fail the invariance check when the current set of formulas is assumed to hold of the current-state variables (according to Definition 4). This can be accomplished by computing the least subset of the

$\mathcal{B}_{\ell,\varphi}$ variables to disable, given the current subset of the $\mathcal{A}_{\ell,\varphi}$ variables, where the minimality ensures that only the formulas that fail the invariance check are removed. We use *Minimal Unsatisfiable Subset* (MUS) to denote such a subset.

Suppose we are interested in computing the minimal subset of a set V of literals such that, together with another fixed set T of literals, the constraints in $\mathcal{C}$ are unsatisfiable. There are several choices for implementing such a MUS computation. A naïve approach is to simply call $\textsc{Sat}(\mathcal{C}, T \cup V)$ and hope that the *unsat core* returned by the SAT/SMT solver is minimal. However, in our particular case, this is guaranteed to fail for the following reason. The set V corresponds to $\{\neg\mathcal{B}_{\ell,\varphi}\}_{\ell,\varphi}$ and the DPLL-style search strategy employed by present day SAT/SMT solvers works by first setting all the assumption literals to true. Given that setting all these assumption literals to true makes the constraint in (6) unsatisfiable, the solver immediately deduces $\perp$ and returns the entire set as the unsat core. For this reason, we use an alternative MUS computation using the routine Mus in Fig. 3.12, which employs a bottom-up iterative strategy. However, note that the minimality of the output of this routine depends on the minimality of the SAT assignments obtained on line 10 in the figure. That is, it is possible that a literal from V assigned to false by the model on line 10 is actually a *dont-care*.

Going back to the routine ExtractInvsImpl, M on line 4 corresponds to the cumulative set of formulas that fail the invariance check and X, on line 5, corresponds to all the other formulas.

Pba finds a subset of assumptions $\hat{\Sigma}_1 \subseteq \Sigma$ such that $\Big(U(\hat{\Sigma}_1, \mathit{bvals})\Big)_{\mathcal{I}}$ is safe with the same proof π of the current under-approximation. As above, we first add the

```
ExtractInvsImpl(C, {A_{ℓ,φ}}_{ℓ,φ}, {B_{ℓ,φ}}_{ℓ,φ})
1   M := ∅, X := {A_{ℓ,φ}}_{ℓ,φ}, Y := {¬B_{ℓ,φ}}_{ℓ,φ}
2   T := X
3   while (S := Mus(C, T, Y)) ≠ ∅ do
4       M := M ∪ S, Y := Y \ M
5       X := {A_{ℓ,φ} | ¬B_{ℓ,φ} ∈ Y}
6       T := X ∪ M
7   return X

Mus(C, T, V)
8   R := ∅
9   while Sat(C, T ∪ R) do
10      m := GetModel(C, T ∪ R)
11      R := R ∪ {v ∈ V | m[v] = false}
12  return R
```

Figure 3.12: Our implementation of ExtractInvs of Fig. 3.7.

constraints under *Lemmas* in Fig. 3.11 to $\mathcal{C}$ which encodes the formulas given by π over current and next-state variables guarded by fresh Boolean variables. Then, the constraints under *Bound Vals* in Fig. 3.11 are added to $\mathcal{C}$ to encode the under-approximation. This reduces the check for whether the map π is a safety proof to that of unsatisfiability of a formula. Finally, $\textsc{Sat}(\mathcal{C}, \Sigma \cup \{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi} \cup \{\mathcal{B}_{\ell,\varphi}\}_{\ell,\varphi})$ is invoked. As π is a safety proof of the under-approximation, this must result in an unsat core. Projecting the core onto Σ gives us the desired $\hat{\Sigma}_1 \subseteq \Sigma$ which identifies the new abstraction and, together with the current *bvals*, the corresponding new under-approximation. The minimality of $\hat{\Sigma}_1$ depends on the algorithm for extracting an unsat core, which is part of the SMT engine of Z3 in our case. In practice, we make iterative SAT calls with the current subset of Σ in place of Σ, until the returned unsat core is the same as the previous subset of assumptions. Note that, as we treat $\{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi}$ and $\{\mathcal{B}_{\ell,\varphi}\}_{\ell,\varphi}$ as assumption literals as well, the SAT/SMT solver

can ignore any redundant formulas in the proof π and such redundancy is quite possible in practice.

NextUnder. Given the current valuation *bvals* and the new abstraction $\hat{\Sigma}$, this routine returns $U(\hat{\Sigma}, \lambda b \in B \cdot bvals(b) + 1)$.

Cegar and IsFeasible. Let $\Big(U(\hat{\Sigma}, bvals)\Big)_{\mathcal{I}}$ be unsafe with a counterexample $\mathscr{C}$. We create a new set of constraints $\mathcal{C}_{\mathscr{C}}$ corresponding to the unrolling of $\tau_\Sigma \wedge \tau_B$ along the control path of $\mathscr{C}$ and check $\textsc{Sat}(\mathcal{C}_{\mathscr{C}}, \Sigma)$. If this returns a satisfiable assignment, the counterexample is feasible in $\tilde{P}$ and the assignment is used to find a counterexample to safety in $\tilde{P}$. Otherwise, we obtain an unsat core $\hat{\Sigma}_1 \subseteq \Sigma$ and refine the abstraction to $\hat{\Sigma} \cup \hat{\Sigma}_1$.

We conclude the section with a discussion of the implementation choices. The above implementation of NextUnder increments all bounding variables uniformly. An alternative is to increment the bounds only for the loops for which the formulas in the current proof π fail to be invariant (e.g., [5, 89]). However, we leave the exploration of such strategies for future. Our use of Z3 is sub-optimal as each call to Solve requires constructing a new Horn-SMT problem. This incurs an unnecessary pre-processing overhead that can be eliminated by a tighter integration with Z3. For Pba and ExtractInvs, we use a single SMT-context with a single copy of the transition relation of the program (without unrolling it) by means of the *Global* constraints mentioned above. This SMT-context is preserved across iterations of Spacer. Constraints specific to a routine are added and retracted using the incremental solving API of Z3. This is vital for good performance in practice. For Cegar and IsFeasible, we unroll the transition relation of the program along the

control path of the counterexample trace returned by Z3. We experimented with an alternative implementation that instead validates each individual step of the (symbolic) counterexample using the same global context as PBA. While this made each refinement step faster, it increased the number of refinements, becoming inefficient overall.

3.6 Experimental Results

We have a prototype implementation of SPACER using the SMT solver Z3 [45]. The implementation and complete experimental results are available online.[7]

Benchmarks. We used the C program benchmarks of the Software Verification Competition 2013 [3]. As our tool does not yet handle memory related properties, we confined ourselves to the categories of *systemc*, *product-lines*, *device-drivers-64* and *control-flow-integers*. All the benchmarks are available on the competition website [3]. We give a brief description of the 4 categories below (and refer the reader to the competition website for more details):

systemc: these are derived from SystemC programs in the literature, which have been transformed to pure C programs by incorporating the scheduler into the C code.

product-lines: these are derived from a research project for integration verification of software product lines.

device-drivers-64: these are derived from the Linux Driver Verification [2] project

[7] http://www.cs.cmu.edu/~akomurav/projects/spacer/home.html

and correspond to the actual Linux kernel code.

control-flow-integers: this contains programs whose safety properties depend mostly on the control-flow structure and integer variables, taken from the repositories of the tools BLAST [70] and CPAchecker [1].

As mentioned in the previous section, we used the implementation of the algorithm GPDR [74] in Z3 for the SOLVE step in each iteration of SPACER. The front-end, which translates a C program to the Horn-SMT format of Z3, is based on the tool UFO [8]. The encoding in Horn-SMT only uses the theory of Linear Rational Arithmetic. All experiments were carried out on an Intel® Core™2 Quad CPU of 2.83GHz and 4GB of RAM. The resource limits were set to 15 minutes of time and 2GB of memory.

Overall, there are 1,990 benchmarks (1,591 marked SAFE, and 399 marked UNSAFE); 1,382 are decided by the front-end of UFO that uses common compiler optimizations. This left 608 benchmarks (231 SAFE, and 377 UNSAFE). To evaluate the advantage of abstractions, we also ran (the implementation of GPDR in) Z3 by itself on the benchmarks and compared with SPACER.

For the UNSAFE benchmarks, Fig. 3.13 shows a scatter plot for the 369 benchmarks verified in both settings, with and without abstraction; of the remaining 8 benchmarks, 6 are unverified, and 2 are verified without abstraction but not by SPACER. Note that, even though abstraction did not help for these benchmarks, the properties are easy, with SPACER needing at most 3 minutes each. We will show later in the section that a traditional CEGAR approach (without PBA) can be even worse.

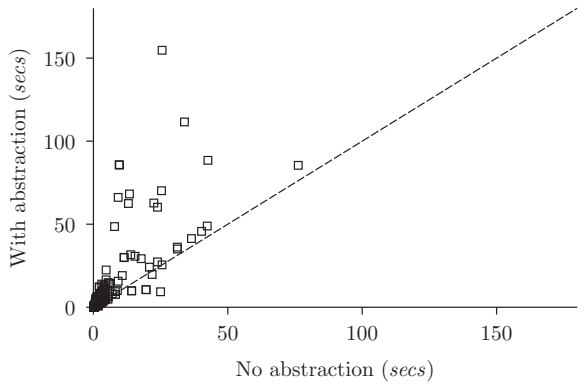


Figure 3.13: Advantage of abstractions (SPACER vs. Z3) for UNSAFE benchmarks.

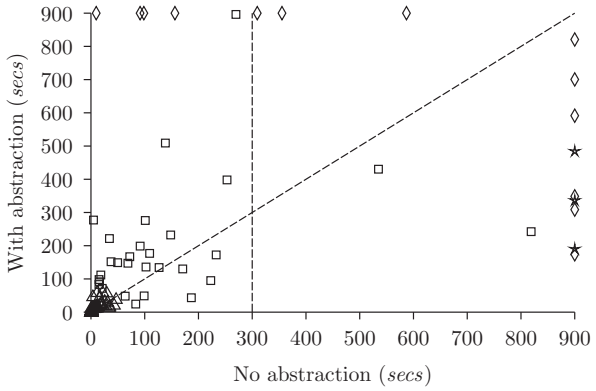


Figure 3.14: Advantage of abstractions (SPACER vs. Z3) for SAFE benchmarks.

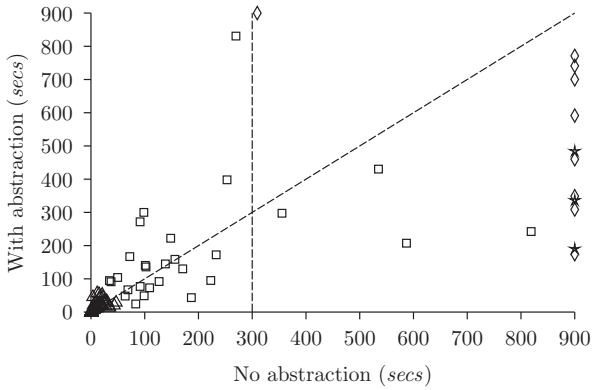


Figure 3.15: The best of the three variants of SPACER against Z3 for SAFE benchmarks.

For the SAFE benchmarks, see Fig. 3.14 for a scatter plot comparing model checking with and without abstraction (i.e., SPACER and Z3, respectively). 176 benchmarks are verified in under a minute in both settings (see the dense set of triangles in the lower left corner of the figure). For these benchmarks, the difference in runtime is not significant enough to be meaningful. We analyze the rest of the results below.

Detailed Results. Table 3.1 shows the experimental results on the 42 safe benchmarks verified by either tool and needing more than a minute of running time. The t columns under Z3 and SPACER show the running times in seconds with 'TO' indicating a time-out and a 'MO' indicating a mem-out. The best times are highlighted in bold. The corresponding scatter plot in Fig. 3.14 shows that the results are mixed for a time bound of 300 seconds (5 minutes). But beyond 5 minutes, abstraction really helps with many benchmarks verified by SPACER when Z3 runs out of time (time-outs are indicated by diamonds and mem-outs are indicated by stars). The couple of benchmarks where SPACER runs out of time become better than Z3 using a different setting, as discussed later. Overall, abstraction helps for *hard* benchmarks. Furthermore, in `elev_13_22`, `elev_13_29` and `elev_13_30`, SPACER is successful even though Z3 runs out of memory, showing a clear advantage of abstraction (this corresponds to the stars in the far right of Fig. 3.14). Note that the `gcnr` example in the table under *misc* is from Fig. 3.1.

The B column in the table shows the final values of the loop bounding variables under the mapping *bvals*, i.e., the maximum number of loop iterations (of any loop)

that was necessary for the final safety proof. Surprisingly, they are very small in many of the hard instances in *systemc* and *product-lines* categories.

Columns a_f and a_m show the sizes of the final and maximal abstractions, respectively, measured in terms of the number of the original constraints used. Note that this only corresponds to the *syntactic* abstraction (see Section 3.4). The final abstraction computed by SPACER is very aggressive. Many constraints are irrelevant (given the computed invariants) with often, more than 50% of the original constraints abstracted away. Finally, the difference between a_f and a_m is insignificant in all of the benchmarks.

An alternative approach to PBA is to restrict the abstraction to state-variables by allowing some of the variables to take next-state values non-deterministically without any constraints, similar to the work by Vizel et al. [105] in the context of hardware verification. This was especially effective for *ssh* and *ssh-simplified* categories – see the entries marked with '*' under column t.

An alternative implementation of CEGAR is to concretize the under-approximation (by refining $\hat{\Sigma}$ to Σ) whenever a spurious counterexample is found. This is analogous to Proof-Based Abstraction (PBA) [91] in hardware verification. Run-time for PBA and the corresponding final values of the bounding variables are shown in columns t_p and B_p of Table 3.1, respectively. While this results in more time-outs, it is significantly better in 14 cases (see the entries marked with '†' under column t_p), with 6 of them comparable to Z3 and 2 (viz., `toy` and `elev_1_31`) significantly better than Z3.

Benchmark	Z3	SPACER					
	t (*sec*)	t (*sec*)	B	a_f (%)	a_m (%)	t_p (*sec*)	B_p
systemc							
pipeline	224	**120**	4	33	33	249	4
tk_ring_06	64	**48**	2	59	59	65	2
tk_ring_07	69	120	2	59	59	†**67**	2
tk_ring_08	232	**158**	2	57	57	358	2
tk_ring_09	817	**241**	2	59	59	266	2
mem_slave_1	536	**430**	3	24	34	483	2
toy	TO	822	4	32	44	†**460**	4
pc_sfifo_2	**73**	137	2	41	41	TO	–
product-lines							
elev_13_21	TO	**174**	2	7	7	TO	–
elev_13_22	MO	**336**	2	9	9	624	4
elev_13_23	TO	**309**	4	6	14	TO	–
elev_13_24	TO	**591**	4	9	9	TO	–
elev_13_29	MO	**190**	2	6	10	TO	–
elev_13_30	MO	**484**	3	11	13	TO	–
elev_13_31	TO	**349**	4	8	17	TO	–
elev_13_32	TO	**700**	4	9	9	TO	–
elev_1_21	**102**	136	11	61	61	161	11
elev_1_23	**101**	276	11	61	61	†140	11
elev_1_29	92	199	11	61	62	†**77**	11
elev_1_31	127	135	11	62	62	†**92**	11
elev_2_29	**18**	112	11	56	56	†26	11
elev_2_31	**16**	91	11	57	57	†22	11
ssh							
s3_clnt_3	109	∗90	12	13	13	**73**	12
s3_srvr_1	187	**43**	9	18	18	661	25
s3_srvr_2	587	∗**207**	14	3	7	446	15
s3_srvr_8	99	**49**	13	18	18	TO	–
s3_srvr_10	83	**24**	9	17	17	412	21
s3_srvr_13	355	∗**298**	15	8	8	461	15
s3_clnt_2	**34**	∗124	13	13	13	†95	13
s3_srvr_12	**21**	∗64	13	8	8	54	13
s3_srvr_14	**37**	∗141	17	8	8	†91	17
s3_srvr_6	**98**	TO	–	–	–	†300	25
s3_srvr_11	**270**	896	15	14	18	831	13
s3_srvr_15	**309**	TO	–	–	–	TO	–
s3_srvr_16	**156**	∗263	21	8	8	†159	21
ssh-simplified							
s3_srvr_3	171	130	11	21	21	**116**	12
s3_clnt_3	**50**	∗139	12	17	22	†104	13
s3_clnt_4	**15**	∗76	12	22	22	56	13
s3_clnt_2	**138**	509	13	26	26	†145	13
s3_srvr_2	**148**	232	12	16	23	222	15
s3_srvr_6	**91**	TO	–	–	–	†272	25
s3_srvr_7	**253**	398	10	20	26	764	10
misc							
gcnr	TO	56	26	81	95	**50**	25

Table 3.1: Comparison of Z3 and SPACER. t and t_p are running times in seconds; B and B_p are the final values of the bounding variables; a_f and a_m are the fractions of assumption variables in the final and maximal abstractions, respectively.

See Fig. 3.15 for a scatter plot using the best running times for SPACER of all the three variants described above.

Advantage of PBA. To better understand the effect of Proof-Based Abstraction (PBA), we ran SPACER with PBA disabled and choosing the coarsest abstraction as our initial abstraction. Note that this is the traditional CEGAR approach. Fig. 3.16 and 3.17 show the scatter plots for the same benchmarks as above comparing CEGAR with and without PBA (PBA + CEGAR is essentially SPACER as described until now). These plots show that, in many cases, PBA results in quite a significant improvement over traditional CEGAR-based abstraction refinement. We believe that this is because PBA results in abstractions that are relevant (proof-based) and precise (due to invariants). The runtimes shown in the plot correspond to the best of the abstraction mechanisms with and without restricting to the state-variables as mentioned above.

Table 3.2 also shows the number of abstraction refinement iterations and the final abstraction size with and without PBA on the 8 safe examples for which both approaches terminate and CEGAR takes more than 5 minutes. Despite the fact that adding PBA results in seemingly more amount of work (for invariant extraction, re-abstraction after each iteration of SPACER, etc.) to maintain coarser abstractions (compare columns a_m and a), the number of abstraction refinement iterations is increased only in a couple of examples. Interestingly, the value of the bounding variables for which both approaches terminate is the same for all the examples.

We conclude this section by comparing our results with UFO [8] — the win-

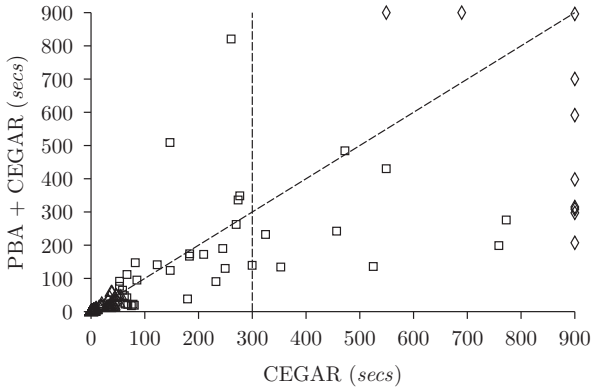


Figure 3.16: Advantage of PBA for SAFE benchmarks.

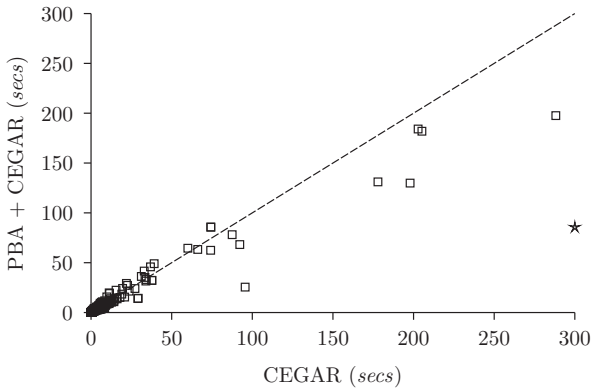


Figure 3.17: Advantage of PBA for UNSAFE benchmarks.

ner of the 4 benchmark categories at SV-COMP'13. The competition version of UFO runs several engines in parallel, including engines based on Abstract Interpretation, Predicate Abstraction, and SMT-based model checking with Interpolation. UFO outperforms SPACER and Z3 in *ssh* and *product-lines* categories by an order of magnitude. These benchmarks seem to be easier for Abstract Interpretation and Predicate Abstraction used in UFO but this needs more investigation. Even so, note that SPACER finds really small abstractions for these categories upon termination.

Benchmark	PBA + CEGAR					CEGAR			
	t (*sec*)	B	$\#iters$	a_f (%)	a_m (%)	t (*sec*)	B	$\#iters$	a (%)
systemc									
`tk_ring_09`	242	2	**11**	59	59	457	2	12	59
`mem_slave_1`	430	3	59	24	**34**	549	3	**55**	36
product-lines									
`elev_13_30`	484	3	**60**	11	**13**	472	3	65	14
`elev_1_21`	136	11	**17**	61	**61**	525	11	18	64
`elev_1_23`	276	11	**16**	61	**61**	772	11	17	63
`elev_1_29`	199	11	17	61	62	759	11	17	62
`elev_1_31`	134	11	17	62	**62**	353	11	17	63
ssh-simplified									
`s3_srvr_2`	232	12	22	12	**16**	*592	12	**17**	22

Table 3.2: Analyzing the effect of PBA on some hard examples. t denotes the running time; B is the final value of the bounding variables; $\#iters$ denotes the number of abstraction refinement iterations; a_f and a_m are the fractions of assumption variables in the final and maximal abstractions with PBA; a is the fraction of the assumption variables in the final abstraction (which is also the maximal) without PBA.

However, in the *systemc* category both SPACER and Z3 perform better than UFO by verifying hard instances (e.g., `tk_ring_08` and `tk_ring_09`) that are not verified by any tool in the competition. Moreover, SPACER is faster than Z3, in general, as shown above. Thus, while SPACER itself is not the best tool for all benchmarks, it is a valuable addition to the state-of-the-art.

3.7 Related work

The most prominent approach for iteratively checking bounded safety is to combine BMC with Craig Interpolation [5, 88, 89]. Recently, algorithms for *incremental* BMC, together with interpolation, have also been proposed [25, 34, 48, 74]. Although our implementation uses Z3 (for SOLVE), which is based on an incremental algorithm, it can be implemented on top of any interpolation-based solver.

Proof-based Abstraction (PBA) was first introduced in hardware verification to

leverage the power of SAT-solvers to focus on relevant facts [66, 91]. Over the years, it has been combined with CEGAR [12, 13], interpolation [13, 85], and PDR[8] [75], all in the context of hardware model checking. To the best of our knowledge, SPACER is the first application of PBA for automatic abstraction refinement in software model checking.

Our extraction of *maximal invariant subsets* from candidate proofs (of bounded safety, in our case) is similar to HOUDINI [54] and is used in several other algorithms (e.g., [90]). As in SPACER, Jain et al. have also used program invariants to obtain precise abstractions in the context of predicate abstraction [76].

The work of Vizel et al. [105], in hardware verification, that extends PDR with abstraction is the closest to ours. However, SPACER is not tightly coupled with PDR. Moreover, SPACER allows for a rich space of abstractions, whereas Vizel et al. limit themselves to *state variable abstraction*.

Finally, the tool UFO [5, 6] also uses abstraction, but in an orthogonal way. UFO uses abstraction to guess the depth of unrolling (plus useful invariants), BMC to detect counterexamples, and interpolation to synthesize safe invariants.

3.8 Conclusion

In this chapter, we presented the SPACER algorithm that combines Proof-Based Abstraction (PBA) with CounterExample Guided Abstraction Refinement (CEGAR) for verifying safety properties of sequential programs. To our knowledge, this is the first application of PBA to software verification. Our abstraction technique com-

[8] PDR stands for *Property Directed Reachability*.

bines localization with invariants about the program. It is interesting to explore alternatives for such a *semantic* abstraction.

While our presentation is restricted to non-recursive sequential programs, the technique can be adapted to solving the more general Horn Clause Satisfiability problem and extended to verifying recursive and concurrent programs [63].

We have implemented SPACER using Z3 and, in particular, its GPDR engine. Our implementation is only an early prototype and is not heavily optimized nor it is tightly integrated with Z3. Nonetheless, the experimental results on 4 categories of the 2nd Software Verification Competition show that SPACER improves on both Z3 and the state-of-the-art.

The results presented in this chapter are published as part of the proceedings of CAV 2013 [80].

3.A Transforming a Safety Proof of $\tilde{P}$ to that of P

Let $P = \langle L, \ell^o, \ell^e, V, \tau \rangle$ be the input program and let $\tilde{P}$ be obtained by the transformation described in Section 3.5 which adds code to count the number of loop iterations. The following lemma shows how to translate a safety proof of $\tilde{P}$ to that of P. Given a sentence φ over the signature $\mathcal{S} \cup V \cup C \cup B$, we write $\forall B \geq 0, C \geq 0 \cdot \varphi$ to mean $\forall B \cup C \cdot \left(\left(\bigwedge_{x \in B \cup C} x \geq 0 \right) \implies \varphi \right)$.

Lemma 10. *If $\tilde{\pi}$ is a safety proof of $\tilde{P}$, then $\pi = \lambda \ell \cdot \{\forall B \geq 0, C \geq 0 \cdot \varphi \mid \varphi \in \tilde{\pi}(\ell)\}$ is a safety proof of P.*

Proof. We will first show that π is safe. We have

$$\bigwedge \pi(\ell^e) \equiv \bigwedge_{\varphi \in \tilde{\pi}(\ell^e)} \forall B \geq 0, C \geq 0 \cdot \varphi \equiv \forall B \geq 0, C \geq 0 \cdot \bigwedge_{\varphi \in \tilde{\pi}(\ell^e)} \varphi.$$

Given that $\tilde{\pi}$ is safe, $\bigwedge_{\varphi \in \tilde{\pi}(\ell^e)} \varphi \implies \bot$ and hence, $\bigwedge \pi(\ell^e) \implies \bot$.

We will next show that π is an invariant map. As $\top \implies \bigwedge \tilde{\pi}(\ell^o)$, $\top \implies \varphi$ for every $\varphi \in \tilde{\pi}(\ell^o)$ and hence, $\top \implies \forall B \geq 0, C \geq 0 \cdot \varphi$. So, $\top \implies \bigwedge \pi(\ell^o)$.

Assume an arbitrary $\mathcal{S}$-structure (that is also a model of Th). Let s, s' be a pair of current and next states satisfying $\bigwedge \pi(\ell_i) \wedge \tau(\ell_i, \ell_j)$ for some $\ell_i, \ell_j \in L$. We need to prove that $\forall B \geq 0, C \geq 0 \cdot \varphi$ is true for s', for every $\varphi \in \tilde{\pi}(\ell_j)$. Let b', c' be arbitrary non-negative values for B, C, respectively. One can easily show that $\tau_B(\ell_i, \ell_j)$ is invertible for non-negative values of the post-variables and let b, c be the values of the corresponding pre-variables. Now, for b, c and s, we know that $\bigwedge \tilde{\pi}(\ell_i)$ is true. Given that $\tilde{\pi}$ is a proof of $\tilde{P}$, it follows that φ is true for b', c' and s'. $\square$

Chapter 4

Probabilistic Systems and Simulation

4.1 Introduction

We will now consider safety of systems with probabilistic behavior. As mentioned in Chapter 1, such systems are increasingly used for a variety of applications and it is important to be able to efficiently verify their correctness. In particular, we consider the problem of checking *strong simulation conformance* between two probabilistic transition systems, an implementation and a specification. In this chapter, we will describe the basic definitions and algorithms which are used in later chapters.

We start with defining our notion of a probabilistic transition system in Section 4.2. We will then define the conformance relation we are interested in, along with several key properties of the relation, in Section 4.3. Following that, we will describe several algorithms for monolithic verification of the conformance relation in Section 4.4, which include a new reduction to SMT and a specialized algorithm

for tree-shaped transition systems. As noted in Chapter 1, when the conformance fails to hold between two probabilistic transition systems, there is no existing notion of a diagnostic *counterexample* which explains the failure. However, this turns out be an essential ingredient for automating compositional reasoning as we will see in Chapters 5 and 6. Section 4.5 describes our characterization of a counterexample to simulation conformance, including several key properties and an algorithm for obtaining a counterexample. Finally, we will define the notion of *parallel composition* between transition systems and show the soundness and completeness of the *assume-guarantee* inference rule ASym, also mentioned in Chapter 1.

4.2 Probabilistic Transition Systems

In the probabilistic transition systems we consider, a transition from a state leads to a *discrete probability distribution* over states.[1] Thus, given a finite, non-empty set S of states, a state $s \in S$, and a label a, a *transition* is a triple (s, a, μ) for a discrete probability distribution μ over S. We use $s \xrightarrow{a} \mu$ to denote such a transition and Fig. 4.1 shows an example. We use $\mathit{Dist}(S)$ to denote the set of all discrete probability distributions over S. For $\mu \in \mathit{Dist}(S)$, the *support* of μ, denoted $\mathit{Supp}(\mu)$, is defined to be the subset of S where each state has a non-zero probability under μ, i.e., $\mathit{Supp}(\mu) = \{s \in S \mid \mu(s) > 0\}$. For $X \subseteq S$, $\mu(X)$ stands for $\sum_{s \in X} \mu(s)$. We use δ_s to denote the special *Dirac* distribution on $s \in S$ where $\delta_s(s) = 1$ and $\delta_s(S \setminus \{s\}) = 0$.

[1] To emphasize, one can perhaps use the term *probabilistic transition* but we avoid the adjective for brevity.

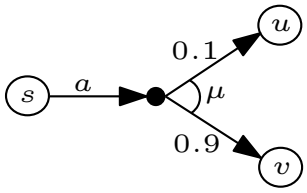


Figure 4.1: An example transition from a state s on an action a to a discrete probability distribution μ over the states u and v.

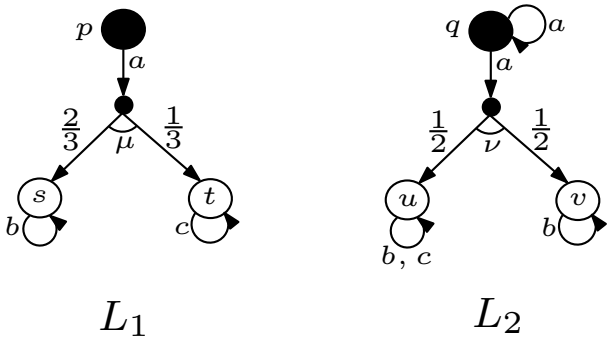


Figure 4.2: Two Labeled Probabilistic Transition Systems L_1 and L_2.

We use the following definition of a transition system to represent probabilistic systems.

Definition 7. *A* Labeled Probabilistic Transition System *(LPTS) is a tuple* $\langle S, s^0, \alpha, \tau \rangle$ *for a finite set of states* S *with a designated start state* s^0*, a finite set of actions* α*, and a finite set of transitions* $\tau \subseteq S \times \alpha \times \text{Dist}(S)$*.*

An LPTS is called reactive *if* τ *is a partial function from* $S \times \alpha$ *to* $\text{Dist}(S)$*, i.e.,* τ *allows at most one transition on a given action from a given state. An LPTS is called* fully-probabilistic *if* τ *is a partial function from* S *to* $\alpha \times \text{Dist}(S)$*, i.e.,* τ *allows at most one transition from a given state.*

For example, Fig. 4.2 shows two example LPTSes L_1 and L_2 where the start states are denoted by filled circles. As the example shows, we allow multiple, possibly non-

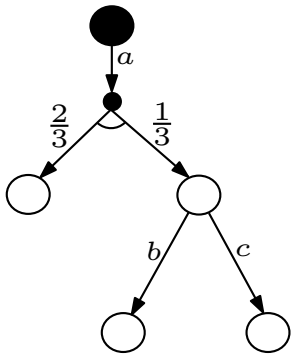


Figure 4.3: An example of a stochastic tree.

deterministic, transitions outgoing from a given state.[2] Note that L_1 is reactive while L_2 is not reactive because of the non-determinism on action a from state q. The figure also shows Dirac distributions on actions b and c.

In the literature, an LPTS is also called a *simple probabilistic automaton* [102]. Similarly, a reactive (fully-probabilistic) LPTS is also called a (Labeled) *Markov Decision Process* (*Markov Chain*). Also, note that an LPTS with all the distributions restricted to Dirac distributions is the classical (non-probabilistic) *Labeled Transition System* (LTS); thus a *reactive* LTS corresponds to the standard notion of a *deterministic* LTS.

We are also interested in LPTSes with a tree structure, i.e., the start state is not in the support of any transition's distribution and every other state is in the support of exactly one transition's distribution. We call such LPTSes *stochastic trees* or simply *trees*. For example, Fig. 4.3 shows a stochastic tree.

[2]This is useful for high level modeling where multiple probabilistic behaviors are allowed. Moreover, the nature of counterexamples to strong simulation (see Section 4.5) and the algorithms for compositional reasoning (see Chapters 5 and 6) do not simplify if non-determinism is disallowed.

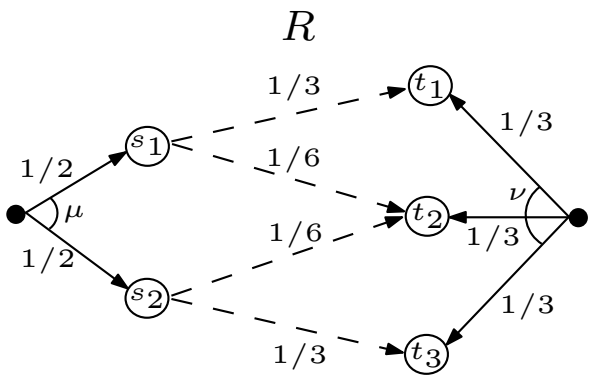


Figure 4.4: Two discrete probability distributions, μ over $S = \{s_1, s_2\}$ and ν over $T = \{t_1, t_2, t_3\}$, and a binary relation R between S and T, shown using dotted arrows, such that $\mu \sqsubseteq_R \nu$. The labeling along the R-edges shows a weight function used to establish the relationship $\sqsubseteq_R$.

4.3 Strong Simulation

To specify correctness of a probabilistic system represented by an LPTS, we use the notion of *strong simulation conformance* with respect to a specification LPTS. This is based on the standard definition of simulation conformance between transition systems (a la Milner [93]). Intuitively, we say that a transition system A is *simulated* by another transition system B if one can exhibit a binary relation between the states of A and B such that from every related state pair, there exist transitions in A and B to states that are also related, whenever the transition in A is feasible. For probabilistic systems, however, a transition leads to a distribution of states and we need an appropriate notion for a *related distribution*. For this, we use the following definition by Segala and Lynch [102]:

Definition 8 ([102])**.** *Let S and T be two non-empty finite sets, $R \subseteq S \times T$, and consider distributions $\mu \in \mathrm{Dist}(S)$ and $\nu \in \mathrm{Dist}(T)$. We say that ν is a* related distribution *of μ with respect to R, denoted $\mu \sqsubseteq_R \nu$, iff there is a* weight function

$w : S \times T \to [0,1]$ *such that*

1. *for every* $s \in S$, $\mu(s) = \sum_{t \in T} w(s,t)$,
2. *for every* $t \in T$, $\nu(t) = \sum_{s \in S} w(s,t)$,
3. *for every* $s \in S, t \in T$, $w(s,t) > 0$ *implies* sRt.

Intuitively, $\mu \sqsubseteq_R \nu$ if the probabilities of states in S under μ can be *distributed* to related states in T under R (as suggested by a suitable weight function) to obtain ν. See Fig. 4.4 for an illustration. Here, $S = \{s_1, s_2\}$, $T = \{t_1, t_2, t_3\}$, μ and ν are the uniform distributions over S and T, and R relates the states connected by dotted arrows. The figure also shows a weight function w by means of a labeling of the R-edges by numbers (all other state pairs are mapped to 0 under w). It is easy to check that w satisfies the conditions in the above definition which shows that $\mu \sqsubseteq_R \nu$. Effectively, w distributes the probability $\mu(s_1) = 1/2$ as $1/3$ to t_1 and $1/6$ to t_2, and the probability $\mu(s_2) = 1/2$ as $1/6$ to t_2 and $1/3$ to t_3, resulting in the distribution ν. Checking $\sqsubseteq_R$ reduces to checking whether the maximum flow in an appropriate network is equal to 1.0 [17]. Note that $\sqsubseteq_R$ is a binary relation between distributions, given R. Using $\sqsubseteq_R$, we can define a Milner-style simulation conformance between LPTSes as follows.

Definition 9 (Strong Simulation [102])**.** *Let* $L_1 = \langle S_1, s_1^0, \alpha_1, \tau_1 \rangle$ *and* $L_2 = \langle S_2, s_2^0, \alpha_2, \tau_2 \rangle$ *be two LPTSes.* $R \subseteq S_1 \times S_2$ *is a* strong simulation *iff for every* $s_1 \in S_1$ *and* $s_2 \in S_2$, *if* $s_1 R s_2$ *then the following holds: for every* $a \in \alpha_1$ *and* $s_1 \xrightarrow{a} \mu_1$ *there is a* $\mu_2 \in \mathrm{Dist}(S_2)$ *with the property that* $s_2 \xrightarrow{a} \mu_2$ *and* $\mu_1 \sqsubseteq_R \mu_2$.

For $s_1 \in S_1$ *and* $s_2 \in S_2$, *we say that* s_2 *strongly simulates* s_1, *denoted* $s_1 \preceq s_2$, *iff there is a strong simulation* T *such that* $s_1 T s_2$. L_2 *strongly simulates* L_1, *also*

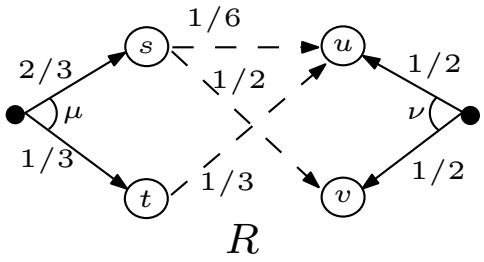


Figure 4.5: Showing $\mu \sqsubseteq_R \nu$ for distributions μ and ν of Fig. 4.2, and a binary relation R shown using dotted arrows. The labeling on the R-edges denotes the weight function used to show $\sqsubseteq_R$.

denoted $L_1 \preceq L_2$*, iff the start state of* L_2 *strongly simulates the start state of* L_1*, i.e.,* $s_1^0 \preceq s_2^0$.

For example, in Fig. 4.2, $L_1 \preceq L_2$ can be shown using the strong simulation $R = \{(p, q), (s, u), (s, v), (t, u)\}$. In particular, the start states p and q are related by R because the outgoing transition $p \xrightarrow{a} \mu$ is *simulated* by the transition $q \xrightarrow{a} \nu$, i.e., $\mu \sqsubseteq_R \nu$. As shown in Fig. 4.5, the latter can be shown using the weight function $w : S_1 \times S_2 \to [0, 1]$ where $w(s, u) = 1/6$, $w(s, v) = 1/2$, $w(t, u) = 1/3$ (s, t, u, and v are as shown in the figure) and w maps every other state pair to 0. Note that q has non-determinism on action a, however, $\mu \not\sqsubseteq_R \delta_q$ as neither s nor t is related to q and hence, there is no corresponding weight function that satisfies the conditions in Definition 8.

Note that $\preceq$ in the above definition can also be seen as a binary relation as follows. When considered between two sets of states S_1 and S_2, $\preceq = \{(s_1, s_2) \in S_1 \times S_2 \mid \exists R \subseteq S_1 \times S_2 \cdot R$ is a strong simulation and $s_1 R s_2\}$. When considered between LPTSes, $\preceq = \{(L_1, L_2) \mid L_2$ *strongly simulates* $L_1\}$. When $L_1 \preceq L_2$, intuitively, L_1 is an *implementation* of the *specification* L_2. The verification problem we are interested in is to check whether the relationship $\preceq$ holds between two LPTSes.

4.3.1 Properties of Strong Simulation

The relations $\sqsubseteq$ and $\preceq$ have some interesting and useful properties which we will describe here.

Let S and T be two non-empty finite sets, $\mu \in \mathit{Dist}(S)$, $\nu \in \mathit{Dist}(T)$, and $R \subseteq S \times T$ such that $\mu \sqsubseteq_R \nu$. Viewing S and T as the two partite sets of a bipartite graph where edges correspond to R, we obtain the following weighted analog of Hall's Marriage Theorem.

Lemma 11 ([106]). *$\mu \sqsubseteq_R \nu$ iff for every $X \subseteq \mathrm{Supp}(\mu)$, $\mu(X) \leq \nu(R(X))$.*

Analogous to strong simulation between non-probabilistic labeled transition systems [93], we have the following properties of $\preceq$.

Lemma 12. *Let L_1 and L_2 be two LPTSes with S_1 and S_2 as the sets of states, respectively. Then, $\preceq \subseteq S_1 \times S_2$ is the coarsest strong simulation between L_1 and L_2, i.e., $\preceq$ is a strong simulation and contains every strong simulation.*

Proof. By Definition 9, $\preceq$ is the union of all strong simulations and hence, contains every strong simulation. To show that $\preceq$ is a strong simulation, it suffices to show that the union of two strong simulations is a strong simulation. The latter is easy to show and we skip the proof. □

Lemma 13 ([102]). *The relation $\preceq$ between LPTSes is a preorder, i.e., reflexive and transitive.*

Proof. Reflexivity, i.e., $L \preceq L$ for an arbitrary LPTS L, can be easily proved by showing that the identity relation is a strong simulation. So, we only consider transitivity here.

Let $L_1 = \langle S_1, s_1^0, \alpha_1, \tau_1 \rangle$, $L_2 = \langle S_2, s_2^0, \alpha_2, \tau_2 \rangle$, and $L_3 = \langle S_3, s_3^0, \alpha_3, \tau_3 \rangle$ be 3 LPTSes with $L_1 \preceq L_2$ and $L_2 \preceq L_3$. Thus, there exist strong simulations $R_{12} \subseteq S_1 \times S_2$ and $R_{23} \subseteq S_2 \times S_3$. We show that the relation $R = R_{23} \circ R_{12}$ is a strong simulation between L_1 and L_3. Let $s_1 R s_3$ and $s_1 \xrightarrow{a} \mu_1$. So, there exists $s_2 \in S_2$ such that $s_1 R_{12} s_2$ and $s_2 R_{23} s_3$. As R_{12} is a strong simulation, there exists $s_2 \xrightarrow{a} \mu_2$ with $\mu_1 \sqsubseteq_{R_{12}} \mu_2$. Similarly, as R_{23} is a strong simulation, there exists $s_3 \xrightarrow{a} \mu_3$ with $\mu_2 \sqsubseteq_{R_{23}} \mu_3$. It suffices to show that $\mu_1 \sqsubseteq_R \mu_3$.

Let $S \subseteq \mathit{Supp}(\mu_1)$ be arbitrary. By Lemma 11, we have $\mu_1(S) \leq \mu_2(R_{12}(S)) \leq \mu_3(R_{23}(R_{12}(S))) = \mu_3(R(S))$. Thus, $\mu_1 \sqsubseteq_R \mu_3$ and hence, R is a strong simulation.

As $s_1^0 R s_3^0$, we conclude that $L_1 \preceq L_3$. □

Finally, we find the following characterization of $\preceq$ useful in the algorithms we will discuss later on.

Lemma 14. *Let $L_1 = \langle S_1, s_1^0, \alpha_1, \tau_1 \rangle$ be a tree and $L_2 = \langle S_2, s_2^0, \alpha_2, \tau_2 \rangle$ be an arbitrary LPTS. Let $R \subseteq S_1 \times S_2$ be such that for every $s_1 \in S_1$ and $s_2 \in S_2$, $s_1 R s_2$ iff the following holds: for every $a \in \alpha_1$ and $s_1 \xrightarrow{a} \mu_1$, there is a $\mu_2 \in \mathrm{Dist}(S_2)$ with the property that $s_2 \xrightarrow{a} \mu_2$ and $\mu_1 \sqsubseteq_R \mu_2$. Then, $R = \preceq$, i.e., $s_1 R s_2$ iff $s_1 \preceq s_2$.*

Proof. It suffices to show that $R \subseteq \preceq$ and $\preceq \subseteq R$. The first direction easily follows from Lemma 12 as R is clearly a strong simulation.

To prove the other direction, we first define the *height* of a state $s \in S_1$ recursively as follows: the height of a leaf state is defined to be 0 and the height of any other state is defined to be one more than the maximum height of any state in the support of any outgoing distribution from that state.

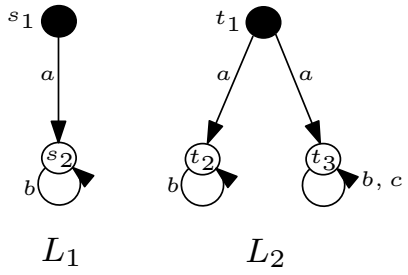


Figure 4.6: An example showing that Lemma 14 does not hold, in general, if L_1 is not a tree. $R = \{(s_1, t_1), (s_2, t_2)\}$ satisfies the definition in the lemma, but $R \subsetneq \preceq$ as $\preceq = \{(s_1, t_1), (s_2, t_2), (s_2, t_3)\}$.

Now, let $s_1 \preceq s_2$. We show that $s_1 R s_2$ by induction on the height of s_1.

For the base case, let s_1 be any leaf state. As s_1 has no outgoing transitions, $s_1 R s_2$ trivially holds by the definition of R.

For the inductive case, let the height of s_1 be non-zero and let $s_1 \xrightarrow{a} \mu_1$. Then, as $\preceq$ is a strong simulation (Lemma 12), there exists μ_2 with $s_2 \xrightarrow{a} \mu_2$ such that $\mu_1 \sqsubseteq_{\preceq} \mu_2$. Let $S \subseteq \mathit{Supp}(\mu_1)$. Then, by Lemma 11, we have $\mu_1(S) \leq \mu_2(\preceq(S))$. As every state in $\mathit{Supp}(\mu_1)$, and hence in S, has a smaller height than that of s_1, by induction hypothesis, $\preceq(S) \subseteq R(S)$ and therefore, $\mu_1(S) \leq \mu_2(R(S))$. As S is arbitrary, we conclude by Lemma 11 that $\mu_1 \sqsubseteq_R \mu_2$. By the definition of R, we obtain that $s_1 R s_2$.

Thus, by induction, we have shown that $\preceq \subseteq R$. □

Note that the condition on R in the lemma is stronger than the one to make it a strong simulation (Definition 9). Also, in general, if L_1 is not a tree, we can only conclude that $R \subseteq \preceq$. See Fig. 4.6 for an example where $R \subsetneq \preceq$.

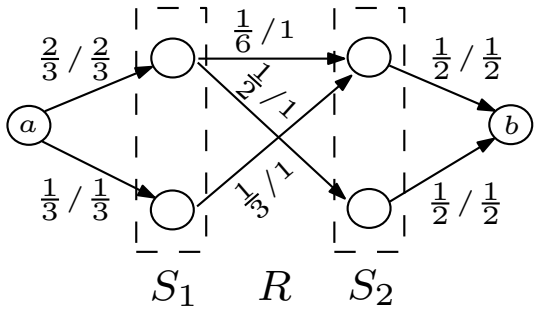


Figure 4.7: The flow network, along with a maximum flow from a to b, to show $\mu \sqsubseteq_R \nu$ where $\mu \in \mathit{Dist}(S_1)$, $\nu \in \mathit{Dist}(S_2)$, and the binary relation $R \subseteq S_1 \times S_2$ are as in Fig. 4.5.

4.4 Algorithms for Strong Simulation

Strong simulation is efficiently decidable in polynomial time and we will describe several algorithms for the problem for several settings.

Checking $\sqsubseteq$. Let S and T be non-empty finite sets and let $\mu \in \mathit{Dist}(S)$ and $\nu \in \mathit{Dist}(T)$. Given $R \subseteq S \times T$, one can check whether $\mu \sqsubseteq_R \nu$ holds by reducing it to a maximum flow computation problem as follows. Consider the graph $F_{\mu,R,\nu} = (S \cup T \cup \{a, b\}, R \cup (\{a\} \times S) \cup (T \times \{b\}))$ denoting a flow network with a and b as the source and the sink nodes. The edges in $F_{\mu,R,\nu}$ are assigned weights according to the function δ as follows: $\delta(a, s) = \mu(s)$ for $s \in S$, $\delta(s, t) = 1$ for sRt, and $\delta(t, b) = \nu(t)$ for $t \in T$. It can then be shown that $F_{\mu,R,\nu}$ has a maximum flow of 1 from a to b iff $\mu \sqsubseteq_R \nu$ [17] which also gives an algorithm for checking $\sqsubseteq_R$.

For example, Fig. 4.7 shows the flow network for distributions μ and ν from Fig. 4.5 as well as a maximum flow function.

Checking $\preceq$. Given two LPTSes $L_1 = \langle S_1, s_1^0, \alpha_1, \tau_1 \rangle$ and $L_2 = \langle S_2, s_2^0, \alpha_2, \tau_2 \rangle$, one can check whether $L_1 \preceq L_2$ holds with a greatest fixed point algorithm that computes the coarsest strong simulation between the LPTSes [17]. Fig. 4.8 shows the pseudo-

code of the algorithm. The algorithm maintains a candidate relation $R \subseteq S_1 \times S_2$, where S_1 and S_2 are the sets of states of the two LPTSes, and iteratively removes pairs from R that violate the condition in Definition 9. The algorithm terminates when a fixed point is reached and returns the coarsest strong simulation between L_1 and L_2. One can then check $L_1 \preceq L_2$ by simply examining if the pair of start states (s_1^0, s_2^0) belongs to the relation returned by the algorithm. If $n = \max(|S_1|, |S_2|)$ and $m = \max(|\tau_1|, |\tau_2|)$, this algorithm takes $O((mn^6 + m^2n^3)/\log n)$ time and $O(mn + n^2)$ space in the worst-case when the candidate relation R is implemented as a queue [17]. There exist several optimizations to this basic algorithm in the literature [106].

Reducing $\preceq$ to SMT. When all the probabilities involved are rational, we can also reduce simulation conformance to satisfiability modulo linear rational arithmetic (i.e., SMT for the theory of linear rational arithmetic) as follows, to take advantage of the efficient SMT solvers that exist today. Given L_1 and L_2 as above, EncodeSim in Fig. 4.9 shows the top-level constraints for encoding $L_1 \preceq L_2$, where we introduce the Boolean variables R_{s_1,s_2} to denote $s_1 R s_2$ for some strong simulation R and rel_{μ_1,μ_2} to denote $\mu_1 \sqsubseteq_R \mu_2$. Here, AddCons simply adds the argument to the pool of constraints, initialized to the empty set. The constraints added on lines 4 and 7 essentially encode the conditions for $L_1 \preceq L_2$ from Definition 9. We encode the constraints on the rel_{μ_1,μ_2} variables on line 6 using EncodeDistRel in Fig. 4.10, where we introduce rational variables w_{t_1,t_2,μ_1,μ_2}'s to denote the weight function in Definition 8 and the variable S_{μ_1,μ_2} to denote the subset of $Supp(\mu_1)$ witnessing $\mu_1 \not\sqsubseteq_R \mu_2$ according to Lemma 11. The constraints added on lines 3–5 encode the necessary conditions for $\mu_1 \sqsubseteq_R \mu_2$ from Definition 8 and the last constraint encodes

```
ComputeSim(L1 = ⟨S1, s1^0, α1, τ1⟩, L2 = ⟨S2, s2^0, α2, τ2⟩)
    R ← S1 × S2  // initialize with all pairs of states
    while true do
        converged ← true
        for (s1, s2) ∈ R do
            sim ← true
            for every s1 -a-> μ1 do
                sim ← false
                for every s2 -a-> μ2 do
                    if μ1 ⊑_R μ2 then
                        sim ← true
                        break
                if sim = false then
                    break
            if sim = false then
                // s1 ⋠ s2
                R ← R \ {(s1, s2)}
                converged ← false
                break
        if converged then
            // fixed-point reached
            return R
```

Figure 4.8: Greatest fixed-point algorithm for computing the coarsest strong simulation relation between two LPTSes L_1 and L_2.

the (contrapositive of the) sufficient condition from Lemma 11. The set S_{μ_1,μ_2} and its image under R can, in turn, be encoded using auxiliary Boolean variables for the states in S_1 and S_2, but we leave the details to the reader.

The following is immediate.

Lemma 15. *$L_1 \preceq L_2$ iff the constraints resulting from* EncodeSim(L_1, L_2) *are satisfiable.*

Checking conformance for trees. We also consider a specialization of the greatest fixed-point algorithm when L_1 is a tree, which is used during abstraction refinement

EncodeSim($L_1 = \langle S_1, s_1^0, \alpha_1, \tau_1 \rangle, L_2 = \langle S_2, s_2^0, \alpha_2, \tau_2 \rangle$)

1 introduce Boolean variables R_{s_1,s_2} to denote $(s_1, s_2) \in R \subseteq S_1 \times S_2$

2 introduce Boolean variables rel_{μ_1,μ_2} to denote $\mu_1 \sqsubseteq_R \mu_2$

3 **for** *every* $(s_1, s_2) \in S_1 \times S_2$ **do**

4 AddCons($R_{s_1,s_2} \implies \bigwedge_{\{(a,\mu_1)|s_1 \xrightarrow{a} \mu_1\}} \bigvee_{\{\mu_2|s_2 \xrightarrow{a} \mu_2\}} rel_{\mu_1,\mu_2}$)

5 **for** *every* $(\mu_1, \mu_2) \in \text{Dist}(S_1) \times \text{Dist}(S_2)$ **do**

`// only the distributions appearing in the transitions`

6 EncodeDistRel($\mu_1, \mu_2, R, rel_{\mu_1,\mu_2}$)

7 AddCons($R_{s_1^0,s_2^0}$)

Figure 4.9: SMT encoding for $L_1 \preceq L_2$.

EncodeDistRel($\mu_1 \in \boldsymbol{Dist}(S_1), \mu_2 \in \boldsymbol{Dist}(S_2), R, b$)

1 introduce rational variables w_{t_1,t_2} for $(t_1, t_2) \in S_1 \times S_2$

2 let S denote a variable denoting a subset of S_1

3 AddCons($b \implies \bigwedge_{t_1 \in S_1} \mu_1(t_1) = \sum_{t_2 \in S_2} w_{t_1,t_2}$)

4 AddCons($b \implies \bigwedge_{t_2 \in S_2} \mu_2(t_2) = \sum_{t_1 \in S_1} w_{t_1,t_2}$)

5 AddCons($b \implies \bigwedge_{t_1 \in S_1, t_2 \in S_2} w_{t_1,t_2} > 0 \implies R_{t_1,t_2}$)

6 AddCons($\neg b \implies \mu_1(S) > \mu_2(R(S))$)

`// S and R(S) can further be encoded with Boolean variables`

Figure 4.10: SMT encoding for $\mu_1 \sqsubseteq_R \mu_2$. Here, b is a Boolean variable denoting the truth value of $\mu_1 \sqsubseteq_R \mu_2$.

(Sections 6.2 and 6.3). Fig. 4.11 shows the pseudo-code of the algorithm which is based on a bottom-up traversal of L_1. It maintains a candidate relation R, initialized to $S_1 \times S_2$. For every non-leaf state $s_1 \in S_1$ in a bottom-up traversal of L_1, the algorithm iteratively checks if a transition $s_1 \xrightarrow{a} \mu_1$ can be simulated by L_2 and for every $s_2 \in S_2$ that does not have a simulating transition on a, removes the pair (s_1, s_2) from R. Correctness can be shown by induction on the height of a state in S_1 and we leave the details to the reader.

```
ComputeSimTree(L1 = ⟨S1, s1^0, α1, τ1⟩, L2 = ⟨S2, s2^0, α2, τ2⟩)
    R ← S1 × S2  // initialize with all pairs of states
    for every non-leaf s1 ∈ S1 in a bottom-up traversal of L1 do
        for every s1 -a-> μ1 do
            for every s2 ∈ R(s1) do
                sim ← false
                for every s2 -a-> μ2 do
                    if μ1 ⊑_R μ2 then
                        sim ← true
                        break
                if sim = false then
                    R ← R \ {(s1, s2)}
    return R
```

Figure 4.11: Specialized fixed-point algorithm for computing the coarsest strong simulation between L_1 and L_2 when L_1 is a tree.

4.5 Counterexamples to Strong Simulation

We have seen several efficient algorithms in the previous section for deciding strong simulation between two LPTSes. However, our techniques for automatic compositional reasoning are iterative and in order to recover from the cases where strong simulation fails to hold, we also need to characterize the notion of a *counterexample* to the conformance relation. We first define a counterexample using a language-theoretic formulation of strong simulation and then characterize counterexamples as stochastic trees.

Definition 10 (Language of an LPTS)**.** *Given an LPTS L, we define its language, denoted $\mathcal{L}(L)$, as the set of all LPTSes simulated by it, i.e., $\{L' \mid L' \text{ is an LPTS and } L' \preceq L\}$.*

We immediately have the following result.

Lemma 16. *For LPTSes L_1 and L_2, $L_1 \preceq L_2$ iff $\mathcal{L}(L_1) \subseteq \mathcal{L}(L_2)$.*

Proof. We know from Lemma 13 that $\preceq$ is transitive and reflexive. In the above statement, necessity follows from the transitivity of $\preceq$ and sufficiency follows from the reflexivity of $\preceq$ which implies $L_1 \in \mathcal{L}(L_1)$. □

So, a counterexample to strong simulation can be defined as follows.

Definition 11 (Counterexample)**.** *Given LPTSes L_1 and L_2 with $L_1 \not\preceq L_2$, a counterexample is an LPTS C such that $C \in \mathcal{L}(L_1) \setminus \mathcal{L}(L_2)$, i.e. $C \preceq L_1$ but $C \not\preceq L_2$.*

Now, L_1 itself is a trivial choice for C but it does not give any more information than what we had before checking the simulation conformance. So, we are interested in counterexamples with simpler structure which retain the relevant information to witness the failure of the conformance relationship. When the probability distributions are all restricted to Dirac distributions, i.e., when we consider LTSes, a *tree-shaped* LTS is known to be sufficient as a counterexample [32]. Based on a similar intuition, we show that a stochastic tree is sufficient as a counterexample for simulation conformance between arbitrary LPTSes.

Theorem 10. *Given LPTSes $L_1 = \langle S_1, s_1^0, \alpha_1, \tau_1 \rangle$ and $L_2 = \langle S_2, s_2^0, \alpha_2, \tau_2 \rangle$ with $L_1 \not\preceq L_2$, there exists a tree counterexample.*

Proof. For $i \in \{1, 2\}$, let (L_i, s) denote the LPTS which is the same as L_i except that the start state is s instead of s_i^0, i.e., $(L_i, s) = \langle S_i, s, \alpha_i, \tau_i \rangle$.

Consider the greatest fixed point algorithm ComputeSim in Fig. 4.8 for computing the coarsest strong simulation between two LPTSes. Let $R \subseteq S_1 \times S_2$ be the relation maintained by ComputeSim(L_1, L_2). We show that whenever a pair (s_1, s_2) is removed from R, there is a tree T_{12} which is a counterexample to $(L_1, s_1) \preceq (L_2, s_2)$.

As $L_1 \not\preceq L_2$, the pair (s_1^0, s_2^0) is eventually removed from R and it will follow that a tree counterexample to $L_1 \preceq L_2$ exists. We proceed by strong induction on the number of iterations of the outermost while loop of ComputeSim.

In the base case, $R = S_1 \times S_2$ and $\mu_1 \sqsubseteq_R \mu_2$ holds for every $\mu_1 \in \mathit{Dist}(S_1)$ and $\mu_2 \in \mathit{Dist}(S_2)$. So, when (s_1, s_2) is removed from R, it must be the case that there is a transition $s_1 \xrightarrow{a} \mu_1$ such that no transition exists from s_2 on action a. Now, let T_{12} be the tree representing the transition $s_1 \xrightarrow{a} \mu_1$ by creating a new state t_1 for s_1 and a new state t_s for every $s \in \mathit{Supp}(\mu_1)$, i.e., $T_{12} = \langle \{t_1\} \cup \{t_s \mid s \in \mathit{Supp}(\mu_1)\}, t_1, \{a\}, \{(t_1, a, \mu_1^t)\}\rangle$, where $\mu_1^t(t_1) = 0$ and $\mu_1^t(t_s) = \mu_1(s)$ for $s \in \mathit{Supp}(\mu_1)$. It can be easily shown that T_{12} is a counterexample to $(L_1, s_1) \preceq (L_2, s_2)$, i.e., $T_{12} \preceq (L_1, s_1)$ but $T_{12} \not\preceq (L_2, s_2)$.

We will now consider the inductive case and let a new pair (s_1, s_2) be removed from the current R. So, there is a transition $s_1 \xrightarrow{a} \mu_1$ which is simulated by no transition on a from s_2. Let $\Delta = \{\nu \in \mathit{Dist}(S_2) \mid s_2 \xrightarrow{a} \nu\}$. So, we know that $\mu_1 \not\sqsubseteq_R \nu$ for every $\nu \in \Delta$. The case of $\Delta = \emptyset$ is the same as the base case above. So, we assume that Δ is non-empty below.

Let $\nu \in \Delta$. Because $\mu_1 \not\sqsubseteq_R \nu$, there exists a set $S_1^\nu \subseteq \mathit{Supp}(\mu_1)$ such that $\mu_1(S_1^\nu) > \nu(R(S_1^\nu))$ by Lemma 11. Let $S_2^\nu = \mathit{Supp}(\nu) \setminus R(S_1^\nu)$. Now, for every pair $(u, v) \in S_1^\nu \times S_2^\nu$, it follows that $(u, v) \notin R$ and by inductive hypothesis, there exists a tree counterexample $T_{u,v}$ for $(L_1, u) \preceq (L_2, v)$.

We build a tree T_{12} as follows. We describe the construction at a high level and leave the details to the reader. As in the base case, we start with representing the transition $s_1 \xrightarrow{a} \mu_1$ as a tree, say T_0, by creating a new state t_1 for s_1 and a new state

CexDistRel($\mu \in \boldsymbol{Dist}(S_1), \nu \in \boldsymbol{Dist}(S_2), R \subseteq S_1 \times S_2$)
  $f :=$ a maximum-flow function for the flow network $F_{\mu,R,\nu}$ (see Section 4.4)
  find $s_1 \in S_1$ with $\mu(s_1) > \sum_{s_2 \in S_2} f(s_1, s_2)$
  $S_1^\nu = \{s_1\}$
  while $\mu(S_1^\nu) \leq \nu(R(S_1^\nu))$ **do**
    $S_1^\nu := \{s_1 \in S_1 \mid \text{exists } s_2 \in R(S_1^\nu) \text{ with } f(s_1, s_2) > 0\}$
  return S_1^ν

Figure 4.12: Finding $S_1^\nu \subseteq S_1$ such that $\mu(S_1^\nu) > \nu(R(S_1^\nu))$, given $\mu \not\sqsubseteq_R \nu$.

t_s for every $s \in \mathit{Supp}(\mu_1)$, i.e., $T_0 = \langle \{t_1\} \cup \{t_s \mid s \in \mathit{Supp}(\mu_1)\}, t_1, \{a\}, \{(t_1, a, \mu_1^t)\}\rangle$, where $\mu_1^t(t_1) = 0$ and $\mu_1^t(t_s) = \mu_1(s)$ for $s \in \mathit{Supp}(\mu_1)$. Then, for every $(u, v) \in \bigcup_{\nu \in \Delta} (S_1^\nu \times S_2^\nu)$, we *attach* the tree $T_{u,v}$ to the state t_u in T_0, i.e., we merge the start state of $T_{u,v}$ with t_u. We claim that T_{12} so obtained is a counterexample to $(L_1, s_1) \preceq (L_2, s_2)$ which we show below.

First of all, it can be easily shown that $T_{12} \preceq (L_1, s_1)$ as T_{12} is essentially a finite unwinding of (L_1, s_1). So, we will only show that $T_{12} \not\preceq (L_2, s_2)$. Let $\nu \in \Delta$ and let R_{12} be the coarsest strong simulation between T_{12} and (L_2, s_2). Consider the set $S = \{t_u \mid u \in S_1^\nu\}$. Now, by construction, we know that for every $(u, v) \in S_1^\nu \times S_2^\nu$, (T_{12}, t_u) is a counterexample to $(L_1, u) \preceq (L_2, v)$ and in particular, we know that $(T_{12}, t_u) \not\preceq (L_2, v)$. This implies that $(t_u, v) \notin R_{12}$ for every such (u, v). In other words, $R_{12}(S) \subseteq R(S_1^\nu)$. Therefore, $\mu_1^t(S) = \mu_1(S_1^\nu) > \nu(R(S_1^\nu)) \geq \nu(R_{12}(S))$ and by Lemma 11, we conclude that $\mu_1^t \not\sqsubseteq_{R_{12}} \nu$. As $\nu \in \Delta$ is arbitrary, this implies that $(t_1, s_2) \notin R_{12}$ and hence, $T_{12} \not\preceq (L_2, s_2)$. □

See Fig. 4.13 for an illustration of a counterexample. Now, in order to obtain an algorithm for computing a tree counterexample from the proof of Theorem 10 above, it remains to show how to compute a subset $S_1^\nu \subseteq \mathit{Supp}(\mu)$ that acts as a witness for

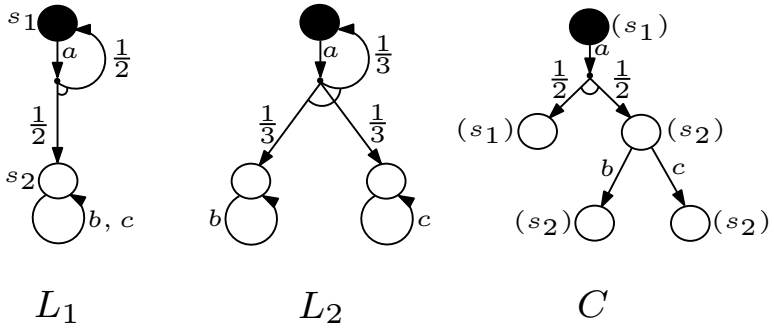


Figure 4.13: C is a counterexample to $L_1 \preceq L_2$.

$\mu \not\sqsubseteq_R \nu$ for distributions $\mu \in \mathit{Dist}(S_1)$, $\nu \in \mathit{Dist}(S_2)$ and $R \subseteq S_1 \times S_2$. See Fig. 4.12 for an algorithm to compute such a witness subset which is analogous to finding a subset failing Hall's condition in Graph Theory and can easily be proved correct.

We have the following complexity result for computing counterexamples.

Theorem 11. *Given LPTSes L_1 and L_2, deciding $L_1 \preceq L_2$ and obtaining a tree counterexample when conformance fails to hold takes $O(mn^6+m^2n^3)$ time and $O(mn+n^2)$ space where $n = \mathtt{max}(|S_{L_1}|, |S_{L_2}|)$ and $m = \mathtt{max}(|\tau_1|, |\tau_2|)$.*

Proof. It can be easily be seen that Algorithm 4.12 takes $O(n^3)$ time and $O(n)$ space which increases the complexity of checking $\mu_1 \sqsubseteq_R \mu_2$ to $O(n^3)$ time and $O(n^2)$ space (see Section 4.4 for an algorithm to decide $\sqsubseteq_R$). The rest of the argument is similar to that of the greatest fixed-point algorithm for computing the coarsest strong simulation [17]. □

Note that the tree counterexample C to $L_1 \preceq L_2$ constructed by the algorithm in the proof of Theorem 10 is essentially a finite *tree execution* of L_1. That is, one can readily obtain a total mapping $M : S_C \to S_1$, where S_C is the set of states of C, with the following property: for every transition $c \xrightarrow{a} \mu_c$ of C, there exists

$M(c) \xrightarrow{a} \mu_1$ such that M is an injection when restricted to $Supp(\mu_c)$ and for every $c' \in Supp(\mu_c)$, $\mu_c(c') = \mu_1(M(c'))$. One can easily show that M is also a strong simulation. We call such a mapping an *execution mapping from C to L_1*. Fig. 4.13 shows an execution mapping in brackets beside the states of C. Note also that, in the inductive case of the proof of the above theorem, *attaching* trees to a state s of C, using the inductive hypothesis, can result in multiple copies of the same transition of L_1 outgoing from s. This can, however, be avoided with additional bookkeeping. This gives us the following corollary, which essentially says that we can always obtain a counterexample without non-determinism as long as L_1 does not have non-determinism.

Corollary 2. *If L_1 is reactive and $L_1 \not\preceq L_2$, there exists a reactive counterexample.*

While Theorem 10 shows that a tree counterexample always exists when simulation conformance fails to hold, it is not immediately clear whether the tree structure (multiple outgoing transitions from a state) is really necessary. The following lemma shows that it is indeed necessary, in general.

Lemma 17. *There exist reactive LPTSes R_1 and R_2 such that $R_1 \not\preceq R_2$ and no counterexample is fully-probabilistic.*

Proof. Consider the two reactive LPTSes R_1 and R_2 in Fig. 4.14. The states, actions, and distributions of the LPTSes are labeled as in the figure. It is easy to see that $r_{11} \not\preceq r_{21}$, $r_{11} \not\preceq r_{23}$, and $r_{11} \preceq r_{22}$. It follows that $\mu_{10}(\{r_{11}\}) = \frac{1}{2} > \mu_{20}(\preceq(\{r_{11}\}) = \mu_{20}(\{r_{22}\}) = \frac{1}{3}$ and hence, $\mu_{10} \not\sqsubseteq_{\preceq} \mu_{20}$ (Lemma 11). Therefore, $r_{10} \not\preceq r_{20}$ and hence, $R_1 \not\preceq R_2$.

Let us assume, for the sake of contradiction, that there is a fully-probabilistic

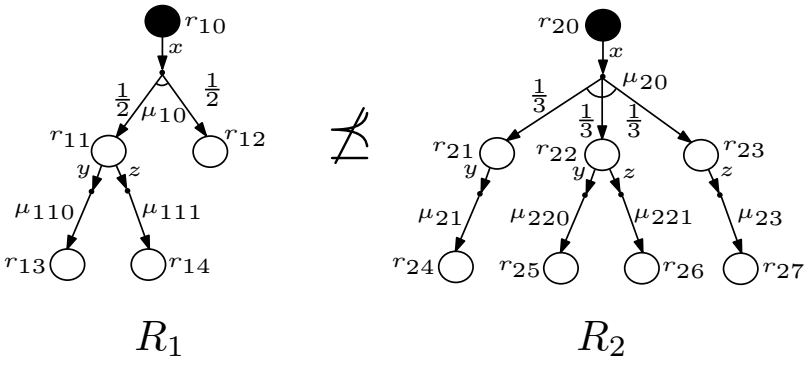


Figure 4.14: An example where there is no fully-probabilistic counterexample.

counterexample C and let its initial state be c_0. Thus, $C \preceq R_1$ but $C \not\preceq R_2$. By Definition 9 there exists a strong simulation U such that $(c_0, r_{10}) \in U$. Given that C is a counterexample and is full-probabilistic, c_0 has exactly one outgoing transition, say $c_0 \xrightarrow{x} \mu_0$. Note that this transition must be labeled by x for it to be simulated by R_1. Let c_1 be an arbitrary state in $Supp(\mu_0)$. If c_1 has any outgoing transitions, it implies that $(c_1, r_{12}) \notin U$. Moreover, in that case, as $\mu_0 \sqsubseteq_U \mu_{10}$, U must include (c_1, r_{11}) and hence, the (only) transition from c_1 must be labeled either y or z. So, let $c_1 \xrightarrow{y} \mu_1$. Now, the only transition on y from r_{11} leads to the distribution μ_{110} and in order to have $\mu_1 \sqsubseteq_U \mu_{110}$, every state in $Supp(\mu_1)$ must be related to r_{13} by U and hence, have no outgoing transitions. One can reach a similar conclusion if the outgoing transition from c_1 is on the action z instead.

To summarize our inferences about C, it must be a tree with exactly one transition from the initial state c_0, say $c_0 \xrightarrow{x} \mu_0$, such that for every state in $Supp(\mu_0)$, there can at most one transition, which can only be labeled either y or z and there are no other transitions. Let S_y and S_z be the sets of states in $Supp(\mu_0)$ with an outgoing transition labeled by y and z, respectively. As $\mu_0 \sqsubseteq_U \mu_{10}$, we have $\mu_0(S_y \cup S_z) \leq$

$\mu_{10}(U(S_y \cup S_z)) = \mu_{10}(\{r_{11}\}) = \frac{1}{2}$, i.e., $\mu_0(S_y \cup S_z) \leq \frac{1}{2}$.

Let V be the smallest binary relation between the states of C and R_2 that satisfies the following conditions. V relates the initial states c_0 and r_{20}. Let c be an arbitrary state of C other than the initial state. If c has no outgoing transitions, V relates c to every state of R_2. If c has its transition labeled by y, V relates it to r_{21} and r_{22}. If c its transition labeled by z, V relates it to r_{22} and r_{23}.

We show that $\mu_0 \sqsubseteq_V \mu_{20}$. Let $X \subseteq \mathit{Supp}(\mu_0)$ be arbitrary. If X includes a state with no transitions, $V(X) = S_2$, the set of all states of R_2 and hence $\mu_0(X) \leq \mu_{20}(V(X)) = 1$. Otherwise, X only has states with transitions labeled by either y or z, i.e., $X \subseteq S_y \cup S_z$, and by the observation made in the above paragraph, $\mu_0(X) \leq \frac{1}{2}$ whereas $\mu_{20}(V(X)) \geq \frac{2}{3}$. Thus, $\mu_0(X) \leq \mu_{20}(V(X))$. This implies that $C \preceq R_2$ which contradicts the assumption that C is a counterexample. □

Thus, even if L_1 does not have non-determinism, i.e., L_1 is reactive, the above theorem shows that a counterexample must have the tree structure (multiple transitions outgoing from a state), in general. This is surprising, as the non-probabilistic counterpart of a fully-probabilistic LPTS is a trace of actions and it is known that trace inclusion coincides with simulation conformance between reactive (i.e., deterministic) LTSes. On a related note, if L_1 is allowed to have non-determinism, one may ask if a reactive LPTS suffices as a counterexample to $L_1 \preceq L_2$. That is not the case either, as the following lemma shows.

Lemma 18. *There exist an LPTS L and a reactive LPTS R such that $L \not\preceq R$ and no counterexample is reactive.*

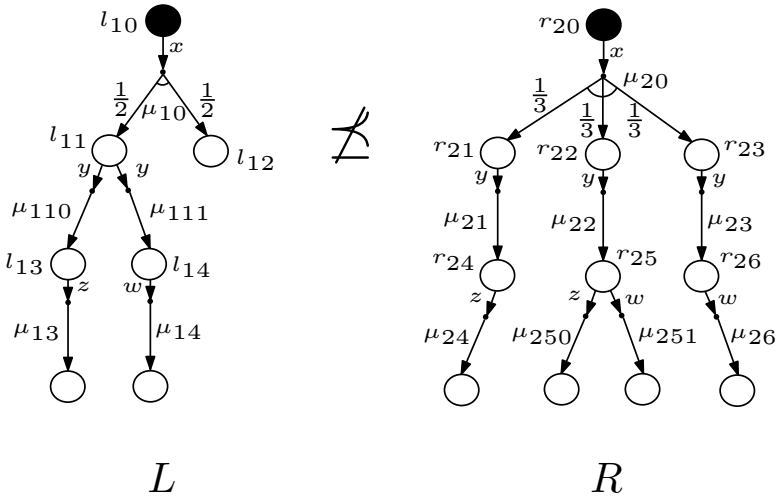


Figure 4.15: There is no reactive counterexample to $L \preceq R$.

Proof. Consider the LPTS L and the reactive LPTS R in Fig. 4.15. The states, actions, and distributions are labeled as in the figure. It is easy to see that $\mu_{110} \not\sqsubseteq_{\preceq} \mu_{23}$ and $\mu_{111} \not\sqsubseteq_{\preceq} \mu_{21}$ whereas $\mu_{110} \sqsubseteq_{\preceq} \mu_{21}, \mu_{22}$ and $\mu_{111} \sqsubseteq_{\preceq} \mu_{22}, \mu_{23}$. It follows that $\mu_{10}(\{l_{11}\}) = \frac{1}{2} > \mu_{20}(\preceq(\{l_{11}\})) = \mu_{20}(\{r_{22}\}) = \frac{1}{3}$ and hence, $\mu_{10} \not\sqsubseteq_{\preceq} \mu_{20}$. Therefore, $l_{10} \not\preceq r_{20}$ and hence, $L \not\preceq R$.

Let us assume, for the sake of contradiction, that there is a reactive counterexample C and let its initial state be c_0. Similar to the arguments made in the proof of Lemma 17, one can show that C must be a tree with exactly one transition from the initial state, say $c_0 \xrightarrow{x} \mu_0$, such that for every state in $Supp(\mu_0)$, there can be at most one transition, which can only be labeled y (because l_{11} has transitions only on y). Let $c_1 \in Supp(\mu_0)$ be such that $c_1 \xrightarrow{y} \mu_1$. It is also the case that all transitions (if any) from a state in $Supp(\mu_1)$ must be labeled by the same action, which can only be either z or w. Let S_y denote the subset of states in $Supp(\mu_0)$ with outgoing transitions (which can only be labeled y). Then, one can also show that

$\mu_0(S_y) \leq \mu_{10}(\{l_{11}\}) = \frac{1}{2}$.

Let V be the smallest binary relation between the states of C and R that satisfies the following conditions. V relates the initial states c_0 and r_{20}. Let c be an arbitrary state of C other than the initial state. If c has no outgoing transitions, V relates c to every state of R. If c has a transition labeled by either z or w, V relates it to all states of R that have transitions labeled z or w, respectively. On the other hand, if c has a transition labeled by y, say $c \xrightarrow{y} \mu$, depending on whether the states in $Supp(\mu)$ have transitions on z, w, or none, V relates c to r_{21} and r_{22}, or r_{22} and r_{23}, or to all three of r_{21}, r_{22} and r_{23}. One can show that V is a strong simulation implying $C \preceq R$. This contradicts the assumption that C is a counterexample. □

4.6 Composition of LPTSes

Parallel composition between LPTSes is defined in the usual way by means of synchronization on common actions. As the transitions in LPTSes are probabilistic, we need the notion of a *product* of two distributions that multiplies the probabilities point-wise, defined as follows. Given two finite sets S and T, and two distributions $\mu \in Dist(S)$ and $\nu \in Dist(T)$, the product of μ and ν, denoted $\mu \otimes \nu$, is a distribution over $S \times T$ such that $(\mu \otimes \nu)(s,t) = \mu(s) \times \nu(t)$ for every $s \in S$ and $t \in T$.

Definition 12 (Composition [102]). *Let $L_1 = \langle S_1, s_1^0, \alpha_1, \tau_1 \rangle$ and $L_2 = \langle S_2, s_2^0, \alpha_2, \tau_2 \rangle$ be two LPTSes. The parallel composition of L_1 and L_2, denoted $L_1 \parallel L_2$, is defined as the LPTS $\langle S, s^0, \alpha, \tau \rangle$, where $S = S_1 \times S_2$, $s^0 = (s_1^0, s_2^0)$, $\alpha = \alpha_1 \cup \alpha_2$, and $((s_1, s_2), a, \mu) \in \tau$ iff one of the following holds:*

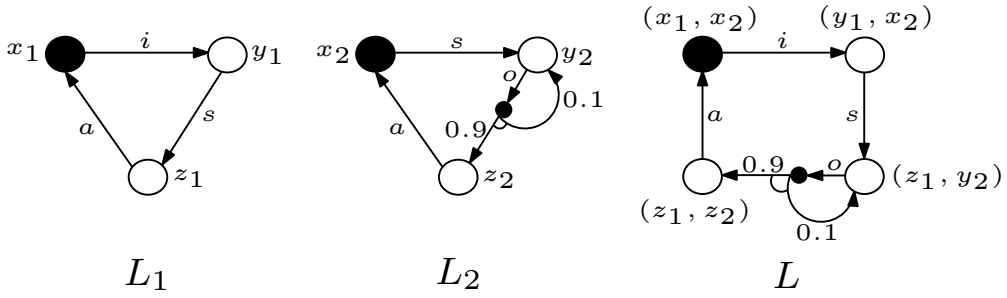


Figure 4.16: Three LPTSes such that L is the parallel composition $L_1 \parallel L_2$.

1. $\mu = \mu_1 \otimes \mu_2$ *for some transitions* $s_1 \xrightarrow{a} \mu_1$ *and* $s_2 \xrightarrow{a} \mu_2$ (a *is common*)
2. $a \notin \alpha_2$ *and* $\mu = \mu_1 \otimes \delta_{s_2}$ *for some transition* $s_1 \xrightarrow{a} \mu_1$ (a *is local to* L_1)
3. $a \notin \alpha_1$ *and* $\mu = \delta_{s_1} \otimes \mu_2$ *for some transition* $s_2 \xrightarrow{a} \mu_2$ (a *is local to* L_2)

For example, in Fig. 4.16, $L = L_1 \parallel L_2$. We have the following useful property.

Lemma 19 ([102])**.** $\preceq$ *is compositional. That is, let* $L_1 = \langle S_1, s_1^0, \alpha_1, \tau_1 \rangle$ *and* $L_2 = \langle S_2, s_2^0, \alpha_2, \tau_2 \rangle$ *be two LPTSes such that* $L_1 \preceq L_2$ *and* $\alpha_2 \subseteq \alpha_1$*. Then, for every LPTS* L*,* $L_1 \parallel L \preceq L_2 \parallel L$*.*

Proof. Let L_1, L_2 and L be as in the statement and let $L = \langle S_L, s_L^0, \alpha_L, \tau_L \rangle$. So, by Definition 9, there exists a strong simulation $R_{12} \subseteq S_1 \times S_2$ such that $(s_1^0, s_2^0) \in R_{12}$. Consider the relation $R = \{((s_1, s), (s_2, s)) \mid s_1 R_{12} s_2 \text{ and } s \in S_L\}$. Clearly, $(s_1^0, s_L^0) R (s_2^0, s_L^0)$. It suffices to show that R is a strong simulation between $L_1 \parallel L$ and $L_2 \parallel L$.

Let $(s_1, s) R (s_2, s)$ and $(s_1, s) \xrightarrow{a} \mu_a$ be arbitrary for some action $a \in \alpha_1 \cup \alpha_L$. By definition of R, $s_1 R_{12} s_2$. By Definition 12, there are three cases to analyze.

$s_1 \xrightarrow{a} \mu_1$, $s \xrightarrow{a} \mu$ and $\mu_a = \mu_1 \otimes \mu$: As $s_1 R_{12} s_2$ and R_{12} is a strong simulation, there exists a transition $s_2 \xrightarrow{a} \mu_2$ with $\mu_1 \sqsubseteq_{R_{12}} \mu_2$. By Definition 12, $(s_2, s) \xrightarrow{a} \mu'_a$

where $\mu'_a = \mu_2 \otimes \mu$. Let $X \subseteq \mathit{Supp}(\mu_a)$. For each $s \in S_L$, define $X^1_s = \{t_1 \mid (t_1, s) \in X\}$. We have

$$\begin{aligned}
\mu_a(X) &= \sum_{s \in S_L} \mu_a(X^1_s \times \{s\}) \\
&= \sum_{s \in S_L} \mu_1(X^1_s) \cdot \mu(s) && \{\text{definition of } \mu_a\} \\
&\leq \sum_{s \in S_L} \mu_2(R_{12}(X^1_s)) \cdot \mu(s) && \{R_{12} \text{ is a strong simulation}\} \\
&= \sum_{s \in S_L} \mu'_a(R_{12}(X^1_s) \times \{s\}) && \{\text{definition of } \mu'_a\} \\
&= \sum_{s \in S_L} \mu'_a(R(X^1_s \times \{s\})) && \{\text{definition of } R\} \\
&= \mu'_a(\bigcup_{s \in S_L} R(X^1_s \times \{s\})) && \{\text{the sets } R(X^1_s \times \{s\}) \text{ are disjoint}\} \\
&= \mu'_a(R(\bigcup_{s \in S_L} X^1_s \times \{s\})) \\
&= \mu'_a(R(X)).
\end{aligned}$$

As X is arbitrary, it follows from Lemma 11 that $\mu_a \sqsubseteq_R \mu'_a$.

$a \notin \alpha_1$, $s \xrightarrow{a} \mu$ and $\mu_a = \delta_{s_1} \otimes \mu$: As $\alpha_2 \subseteq \alpha_1$, $a \notin \alpha_2$ and by Definition 12, $(s_2, s) \xrightarrow{a} \mu'_a$ with $\mu'_a = \delta_{s_2} \otimes \mu$. Now, let $X \subseteq \mathit{Supp}(\mu_a)$ and let $X_2 = \{t \mid (s_1, t) \in X\}$. We have $\mu_a(X) = \mu(X_2) = \mu'_a(\{s_2\} \times X_2) = \mu'_a(R(X))$ and hence, $\mu_a \sqsubseteq_R \mu'_a$.

$s_1 \xrightarrow{a} \mu_1$, $a \notin \alpha_L$ and $\mu_a = \mu_1 \otimes \delta_s$: As $s_1 R_{12} s_2$ and R_{12} is a strong simulation, there exists $s_2 \xrightarrow{a} \mu_2$ with $\mu_1 \sqsubseteq_{R_{12}} \mu_2$. Now, let $X \subseteq \mathit{Supp}(\mu_a)$ and let $X_1 = \{t_1 \mid (t_1, s) \in X\}$. We have $\mu_a(X) = \mu_1(X_1) \leq \mu_2(R_{12}(X_1)) = \mu'_a(R(X))$ and

hence, $\mu_a \sqsubseteq_R \mu'_a$.

Hence, R is a strong simulation. Therefore, $L_1 \parallel L \preceq L_2 \parallel L$. □

Finally, we show the *soundness* and *completeness* of the assume-guarantee inference rule ASYM mentioned in Chapter 1, reproduced below. Here, L_1, L_2, A, and P are all LPTSes.

$$\frac{1 : L_1 \parallel A \preceq P \qquad 2 : L_2 \preceq A}{L_1 \parallel L_2 \preceq P} \text{ (ASYM)}$$

The rule is *sound* if the conclusion holds whenever the premises hold for some assumption LPTS A, and the rule is *complete* if there is an assumption A satisfying the premises whenever the conclusion holds.

Theorem 12. *If $\alpha_A \subseteq \alpha_2$, the rule* ASYM *is sound and complete.*

Proof. Soundness follows from Lemma 19. Completeness follows trivially by replacing A with L_2. □

Chapter 5

Active Learning for Simulation Conformance

5.1 Introduction

We have seen in Chapter 4 that strong simulation conformance between two *Labeled Probabilistic Transition Systems* (LPTSes) is decidable in polynomial time. However, as mentioned in Chapter 1, when an LPTS L is the parallel composition of multiple components, we encounter the *state-space explosion* problem for checking conformance with a specification LPTS P. To address this problem, we follow the assume-guarantee paradigm [98] for compositional reasoning. In particular, we focus on the following assume-guarantee inference rule, which we have shown to be sound and complete (see Chapter 4):

$$\frac{1 : L_1 \parallel A \preceq P \qquad 2 : L_2 \preceq A}{L_1 \parallel L_2 \preceq P} \text{ (ASym)}$$

In other words, in order to show that a probabilistic system composed of two parallel components L_1 and L_2 conforms to a specification P, it suffices to come up with a (preferably small) assumption A about L_2 which can be used in its place to show the conformance together with L_1. In this chapter, we study iterative algorithms for *learning* a small assumption A, given L_1, L_2, and P, from counterexamples to the premises.

In the context of non-probabilistic systems, several algorithms exist for compositional verification that are based on learning the intermediate assumptions from *samples* generated dynamically. In particular, algorithms for compositional verification of trace inclusion [33, 99] and simulation conformance [31] have been studied that are based on learning from *traces* [14, 96] and *trees* [31], respectively. These algorithms are essentially adaptations of *active* learning [14] algorithms for inferring an unknown target system from samples, to the compositional setting. An active learning framework typically has two entities – a *learner* which tries to learn the unknown target and a *teacher* which guides the learner by giving new information in terms of samples. The teacher, typically, can answer two types of queries – *membership* (of a sample in the unknown target) and *equivalence* (between the conjectured model and the unknown target) [14]. The learner terminates when an equivalence query is answered positively by the teacher. In the context of assume-guarantee style

compositional reasoning, the unknown target corresponds to a sufficient assumption in the inference rule and the teacher answers the queries by checking the premises of the rule [99].

However, compositional reasoning for probabilistic systems has not been studied well in the literature. In particular, no algorithms are known (based on learning or otherwise) for compositional verification of simulation conformance. For non-probabilistic systems, simulation conformance between two labeled transition systems (LTSes) reduces to *tree language* inclusion and there exists an adaptation of an active learning algorithm for *deterministic tree automata* to the compositional setting [31]. Now, in the probabilistic setting, we have seen in Chapter 4 that a counterexample to strong simulation between LPTSes is a stochastic tree. So, we can similarly define a *stochastic* tree language such that strong simulation reduces to inclusion between stochastic tree languages. However, while there exist techniques for learning from samples consisting of (non-stochastic) trees with information regarding the probability of acceptance [28], we are not aware of any prior algorithms for learning from stochastic trees. Moreover, we are also not aware of a probabilistic variant of a tree automaton to recognize stochastic tree languages. This motivated us to consider learning an LPTS directly from stochastic tree samples, as opposed to inventing stochastic tree automata and casting the verification problem in automata-theoretic terms.

In our context of active learning, an equivalence query corresponds to asking whether a conjecture C is *strong simulation equivalent* to T, i.e., whether $C \preceq T$ and $T \preceq C$. So, when the equivalence check fails for a conjecture C, a counterexample

```
LearnLPTS()
    P := ∅, N := ∅ // initialize positive and negative tree samples
    while true do
        L := FindConsistent(P, N) // see Section 5.2
        (res, cex) := CheckConjecture(L) // ask teacher
        if res is yes then
            // L is equivalent to target
            return L
        else if res is positive then
            // L does not simulate target, witnessed by cex
            P := P ∪ {cex}
        else if res is negative then
            // target does not simulate L, witnessed by cex
            N := N ∪ {cex}
```

Figure 5.1: Active learning loop for inferring an LPTS using only equivalence queries.

can be of two kinds. If $T \not\preceq C$, then a counterexample tree t satisfies $t \preceq T$ but $t \not\preceq C$ and we call it a *positive* sample. On the other hand, if $C \not\preceq T$, then a counterexample tree t satisfies $t \preceq C$ but $t \not\preceq T$ and we call it a *negative* sample. Now, a membership query would correspond to asking whether a stochastic tree t is simulated by the unknown target LPTS T, i.e., whether $t \preceq T$. However, we observe that such a membership query is challenging to create as the learner would need to guess not only the tree structure but also the transition probabilities. For this reason, we restrict the learning framework such that a teacher can only answer equivalence queries.

Fig. 5.1 shows the pseudo-code of our active learning framework LearnLPTS. The learner maintains a set of *positive* and *negative* tree samples. In each iteration, it infers an LPTS L *consistent* with all the samples, i.e., L simulates all the positive samples and none of the negative samples, and conjectures L to the teacher. If the

teacher finds L to be equivalent to the target, it returns *yes*, and otherwise, it returns a new positive or a new negative sample.

We first describe algorithms for FindConsistent, i.e., for inferring an LPTS consistent with a given set of positive and negative samples, in Section 5.2. Given the ultimate objective of inferring a small assumption in the rule Asym, our main interest is in learning consistent LPTSes of small size. To this end, our algorithms employ two different ways of partitioning the state-space of the counterexamples. We then describe the convergence guarantees of LearnLPTS in Section 5.3. In particular, we show that there is no converging learning algorithm in the presence of an adversarial teacher, but there exists a converging algorithm under a natural assumption on the teacher. We also discuss how convergence is affected when the consistent LPTS in each iteration is required to have the *minimal* number of states. Finally, in Section 5.4, we describe how active learning is adapted for compositional reasoning and discuss the complexity guarantees.

5.2 Learning a Consistent LPTS

Assume that we are given a finite set of *positive* stochastic tree samples, say $\mathcal{P}$, and another finite set of *negative* stochastic tree samples, say $\mathcal{N}$. We are interested in learning an LPTS L such that $P \preceq L$ for every $P \in \mathcal{P}$ and $N \not\preceq L$ for every $N \in \mathcal{N}$. Such an L is said to be *consistent* with the given tree samples. Note that the LPTS obtained by merging the start states of all trees in $\mathcal{P}$, denoted $L_{\mathcal{P}}$, can be easily shown to satisfy $P \preceq L_{\mathcal{P}}$ for every $P \in \mathcal{P}$. If $\mathcal{P} = \emptyset$, we let $L_{\mathcal{P}}$ to be the single-state

LPTS with no transitions. Now, if L is an arbitrary consistent LPTS, one can also show that $L_{\mathcal{P}} \preceq L$ and hence, by Lemma 13, $L_{\mathcal{P}}$ will also be consistent. Thus, one can check, in polynomial time, whether there exists a consistent LPTS by checking $N \preceq L_{\mathcal{P}}$ for every $N \in \mathcal{N}$. However, the size of $L_{\mathcal{P}}$ is equal to the combined size of all trees in $\mathcal{P}$. So, the question we want to address is whether we can find *small* consistent LPTSes.

As mentioned earlier, given our ultimate objective of learning small assumptions for compositional reasoning, we are interested in learning a consistent LPTS of a small size, preferably the smallest. To that effect, we describe algorithms that obtain a *folding* of the tree-shaped $L_{\mathcal{P}}$ into a consistent LPTS. The algorithms we propose draw inspiration from state-space partitioning techniques for obtaining consistent automata from counterexample traces [27, 58, 67, 97]. Let $S_{\mathcal{P}} = \bigcup_{P \in \mathcal{P}} S_P$ and $S_{\mathcal{N}} = \bigcup_{N \in \mathcal{N}} S_N$ where S_L denotes the set of states of an LPTS L. First, we consider an algorithm based on traditional state-space partitioning of $S_{\mathcal{P}}$. While this approach does reduce the number of states in the inferred consistent LPTS, it does not guarantee minimality in terms of the number of states. Nevertheless, as we will see in Section 5.3, we find it useful for the learning loop in LearnLPTS (Fig. 5.1) to converge. We will then introduce a new *stochastic* state-space partitioning which enables us to obtain a minimal consistent LPTS.

5.2.1 Using State-Space Partitioning

We first describe an algorithm based on traditional state-space partitioning of $S_{\mathcal{P}}$. A partition of a set X is a set of non-empty subsets of X such that every element of X

is in exactly one of the subsets. A partition Π of X induces an equivalence relation which relates two elements iff they are in the same subset, i.e., the equivalence classes under the equivalence relation are nothing but the subsets in the partition. For a partition Π of $S_{\mathcal{P}}$ and a state $s \in S_{\mathcal{P}}$, we let $[s]_\Pi$ denote the equivalence class of s. Throughout this section, we assume that the start states of all positive samples ($\mathcal{P}$) are in the same equivalence class, i.e., $[s^0_P]_\Pi = [s^0_Q]_\Pi$ for every $P, Q \in \mathcal{P}$. Given a partition Π, one can obtain a *quotient* LPTS where the states correspond to the equivalence classes and the distributions of the transitions of $\mathcal{P}$ are *lifted* to distributions over equivalence classes:

Definition 13 (Quotient LPTS)**.** *Given a partition* Π *of* $S_{\mathcal{P}}$*, the* quotient *of* $\mathcal{P}$*, denoted* $\mathcal{P}/\Pi$*, is the LPTS* $\langle \Pi, e^0, \alpha, \tau \rangle$ *where* $e^0 = [s^0_P]_\Pi$ *for every* $P \in \mathcal{P}$*,* $\alpha = \bigcup_{P \in \mathcal{P}} \alpha_P$ *and* $(e, a, \mu) \in \tau$ *iff there exists* $(s, a, \mu_p) \in \tau_P$ *for some* $P \in \mathcal{P}$ *with* $[s]_\Pi = e$ *and* μ_p *is lifted to* Π *to obtain* μ*, i.e.,* $\mu(e') = \sum_{s' \in e'} \mu_p(s')$ *for all* $e' \in \Pi$*. We use* $lift_\Pi(\mu_p)$ *to denote the lifting of* μ_p *to the partition* Π*.*

It is straightforward to show that a quotient is a well-defined LPTS, i.e., the liftings of the distributions of $\mathcal{P}$ are well-defined distributions over equivalence classes. The following lemma shows that a quotient simulates every positive sample.

Lemma 20. *Let* Π *be a partition of* $S_{\mathcal{P}}$*. Then,* $P \preceq \mathcal{P}/\Pi$ *for every* $P \in \mathcal{P}$*.*

Proof. Let $P \in \mathcal{P}$ and let $P = \langle S_P, s^0_P, \alpha_P, \tau_P \rangle$. To show that $P \preceq \mathcal{P}/\Pi$, consider the binary relation $R = \{(s, [s]_\Pi) \mid s \in S_P\}$. Note that the start state of $\mathcal{P}/\Pi$ is $[s^0_P]_\Pi$ and hence, the start states of P and $\mathcal{P}/\Pi$ are related by R. It suffices to show that R is a strong simulation.

Let $s \in S_P$ and $s \xrightarrow{a} \mu_p$ be arbitrary. By definition, there exists a transition

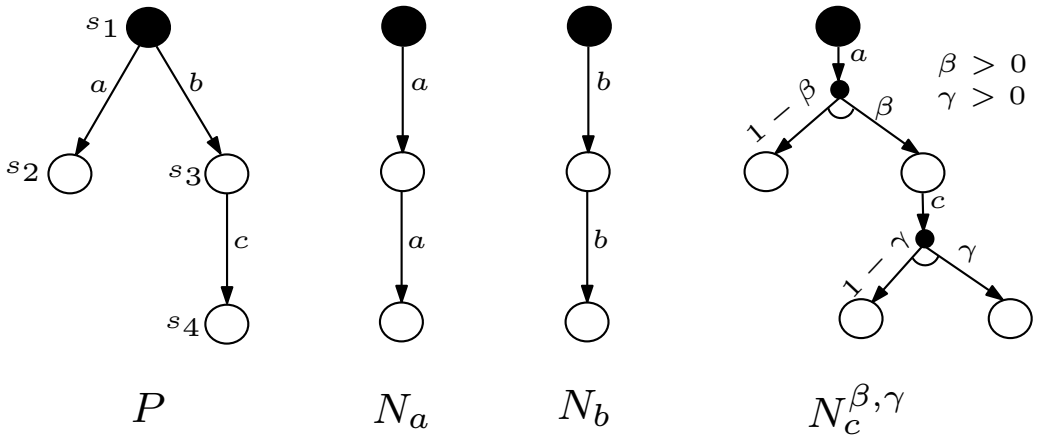


Figure 5.2: Example stochastic trees, divided into a positive sample (P) and 3 negative samples ($N_a, N_b, N_c^{\beta,\gamma}$), for active learning.

$[s]_\Pi \xrightarrow{a} \mu$ in $\mathcal{P}/\Pi$ where $\mu(e) = \mu_p(e)$ for all $e \in \Pi$. It suffices to show that $\mu_p \sqsubseteq_R \mu$. Let $S \subseteq \mathit{Supp}(\mu_p)$. We have,

$$
\begin{aligned}
\mu_p(S) &= \sum_{e \in \Pi} \mu_p(S \cap e) && \{\Pi \text{ is a partition}\} \\
&\leq \sum_{e \in \Pi, S \cap e \neq \emptyset} \mu_p(e) \\
&= \sum_{e \in R(S)} \mu_p(e) && \{\text{definition of } R\} \\
&= \sum_{e \in R(S)} \mu(e) && \{\text{definition of } \mu\} \\
&= \mu(R(S)).
\end{aligned}
$$

As S is arbitrary, it follows from Lemma 11 that $\mu_p \sqsubseteq_R \mu$. We conclude that R is a strong simulation. □

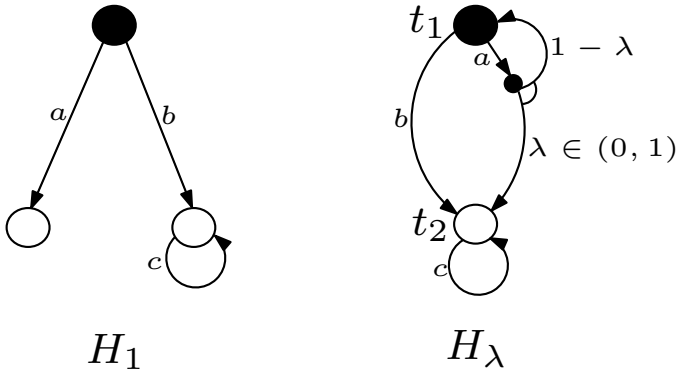


Figure 5.3: Quotients for least size partition (H_1) and stochastic partition (H_λ) of P in Fig. 5.2.

If the quotient LPTS is also consistent, i.e., does not simulate any negative sample, we call the partition a *consistent partition*. For example, Fig. 5.2 shows a positive sample P and 3 negative samples N_a, N_b, and $N^{\beta,\gamma}$ for some $\beta, \gamma \in (0, 1]$. For these samples, H_1 in Fig. 5.3 is a consistent quotient LPTS obtained from the partition $\{\{s_1\}, \{s_2\}, \{s_3, s_4\}\}$ of $S_\mathcal{P}$. The following lemma shows that one can bound the number of states of a consistent LPTS obtained using the partitioning approach.

Lemma 21. *If L is an LPTS of k states that simulates all samples in $\mathcal{P}$, then there exists a partition Π of $S_\mathcal{P}$ of size at most 2^k such that $\mathcal{P}/\Pi \preceq L$.*

Proof. Let $P \in \mathcal{P}$. Let $P = \langle S_P, s_P^0, \alpha_P, \tau_P \rangle$ and $L = \langle S_L, s_L^0, \alpha_L, \tau_L \rangle$. We know that $P \preceq L$. That is, there exists a strong simulation $R_P \subseteq S_P \times S_L$ with $s_P^0 R_P s_L^0$. As P is a tree, s_P^0 is not in the support of any distribution and hence, $R_P(s_P^0)$ does not affect whether R_P is a strong simulation or not. So, without loss of generality, assume that $R_P(s_P^0)$ is the singleton $\{s_L^0\}$. Let $R = \bigcup_{P \in \mathcal{P}} R_P$. Now, R induces an equivalence relation E over $S_\mathcal{P}$ such that $s_1 E s_2$ iff $R(s_1) = R(s_2)$. Let Π be the partition corresponding to E. Note that $[s_P^0]_\Pi = [s_Q^0]_\Pi$ for $P, Q \in \mathcal{P}$, satisfying our

assumption on a partition that the start states of all samples in $\mathcal{P}$ are in the same equivalence class. The size of Π is clearly bounded by 2^k.

In order to show $\mathcal{P}/\Pi \preceq L$, consider the binary relation $R' = \{([s_p]_\Pi, s_l) \mid s_p R s_l, s_p \in S_\mathcal{P}, s_l \in S_L\}$ between the states of $\mathcal{P}/\Pi$ and L. Clearly, R' relates the start state of $\mathcal{P}/\Pi$, say $s^0_{\mathcal{P}/\Pi}$, with s^0_L, as $s^0_P R s^0_L$ for every $P \in \mathcal{P}$. It suffices to show that R' is a strong simulation.

Let $eR's_l$ and $e \xrightarrow{a} \mu$ be arbitrary. By Definition 13, there exists $s_p \in S_\mathcal{P}$ and $\mu_p \in \mathit{Dist}(S_\mathcal{P})$ with $[s_p]_\Pi = e$, $s_p \xrightarrow{a} \mu_p$ and $\mu(e') = \mu_p(e')$ for all $e' \in E$. Furthermore, by the definitions of R' and Π, $s_p R s_l$. As R is a strong simulation, there exists $\mu_l \in \mathit{Dist}(S_L)$ such that $s_l \xrightarrow{a} \mu_l$ and $\mu_p \sqsubseteq_R \mu_l$. Let $E' \subseteq \mathit{Supp}(\mu)$. Now,

$$
\begin{aligned}
\mu(E') &= \sum_{e' \in E'} \mu(e') \\
&= \sum_{e' \in E'} \mu_p(e') \\
&= \sum_{e' \in E'} \mu_p(\{s \in S_\mathcal{P} \mid [s]_\Pi = e'\}) \\
&= \mu_p(\{s \in S_\mathcal{P} \mid [s]_\Pi \in E'\}) \\
&\leq \mu_l(R(\{s \in S_\mathcal{P} \mid [s]_\Pi \in E'\})) && \{\mu_p \sqsubseteq_R \mu_l\} \\
&= \mu_l(\bigcup_{e' \in E'} R(\{s \in S_\mathcal{P} \mid [s]_\Pi = e'\})) \\
&= \mu_l(\bigcup_{e' \in E'} R'(e')) && \{\text{Def. of } R'\} \\
&= \mu_l(R'(E')).
\end{aligned}
$$

As E' is arbitrary, it follows from Lemma 11 that $\mu \sqsubseteq_{R'} \mu_l$. We conclude that R' is a strong simulation. □

Note that, if L and every $P \in \mathcal{P}$ is non-probabilistic (i.e., an LTS), then one can always choose the strong simulation R_P in the above proof to be a function and the bound in the above lemma goes down to k. The following is immediate, using Lemmas 13 and 20.

Corollary 3. *For every consistent LPTS of k states, there is a consistent partition of size at most 2^k.*

In other words, the state-space partitioning approach for learning a consistent LPTS can be at most exponentially worse, in terms of the number of states. While this is only an upper bound, we can also show that this approach cannot guarantee minimality in general. To see this, H_λ in Fig. 5.3, for any $\lambda \in (0, 1)$, is also a consistent LPTS for the samples in Fig. 5.2 with one less state when compared to H_1, the smallest LPTS one can obtain using state-space partitions. This is surprising at first, as it is well known for non-probabilistic systems and trace counterexamples that there always exists a consistent partition of the least number of states [67, 96].

Algorithm

A naïve algorithm for finding a *least-sized consistent partition* is to enumerate all the partitions of $S_{\mathcal{P}}$ for increasing values of the size, and for each of them, check if the corresponding quotient simulates any tree in $\mathcal{N}$. Alternatively, in the case where all the probabilities involved are rational, we can utilize the efficient solvers that exist today for satisfiability modulo theories (SMT), and in particular, for the theory of

```
EncodeConsisPartition(P, N, k)
1   introduce Boolean variables Π_{s,i} to denote [s]_Π = e_i^Π for (s, i) ∈ S_P × {1, ..., k}
2   for s ∈ S_P do
3       AddCons(xor(Π_{s,1}, ..., Π_{s,k}))
4   for P ∈ P do
5       AddCons(Π_{s_P^0,1})
6   for N ∈ N do
        // encode N ⋠ P/Π
7       EncodeNotSim(N, P, Π, k)
```

Figure 5.4: SMT encoding for a consistent partition of size k for samples $\mathcal{P}$ and $\mathcal{N}$.

linear rational arithmetic, as shown below. We expect this to be more efficient than an exhaustive search, in practice. Moreover, this prepares the ground for an optimal algorithm we discuss in the next subsection.

As mentioned at the beginning of the section, we can easily check if there exists a consistent LPTS by merging the start states of all positive samples to obtain $L_{\mathcal{P}}$ and checking if $L_{\mathcal{P}}$ is consistent. If there exists a consistent LPTS, we can search for the smallest consistent partition by iteratively checking if there exists a consistent partition of size k, for increasing values of k. For a given k, we encode the existence of a consistent partition of size k as an SMT problem using EncodeConsisPartition$(\mathcal{P}, \mathcal{N}, k)$ in Fig 5.4. Here, we introduce Boolean variables $\Pi_{s,i}$ to denote $[s]_\Pi = e_i^\Pi$ for some partition $\Pi = \{e_1^\Pi, \ldots, e_k^\Pi\}$ of size k. The constraints added on lines 3 and 5 essentially encode that the partition Π is well-defined, i.e., each state $s \in S_{\mathcal{P}}$ belongs to exactly one element of the partition and the start states of all samples in $\mathcal{P}$ belong to the same equivalence class e_1^Π, respectively.

For Π to be consistent, we need to encode that no sample in $\mathcal{N}$ is simulated by

EncodeNotSim$(N, \mathcal{P}, \Pi, k)$

1 introduce Boolean variables $R_{s_n,i}$ to denote $(s_n, e_i^{\Pi}) \in R \subseteq S_N \times \Pi$

2 introduce Boolean variables rel_{μ_n,μ_p} to denote $\mu_n \sqsubseteq_R lift_{\Pi}(\mu_p)$

3 **for** *every* $(s, i) \in S_N \times \{1, \ldots, k\}$ **do**

4 AddCons$(R_{s_n,i} \iff \bigwedge_{\{(a,\mu_n)|s_n \xrightarrow{a} \mu_n\}} \bigvee_{\{(s_p,\mu_p)|s_p \in S_{\mathcal{P}}, s_p \xrightarrow{a} \mu_p\}} (\Pi_{s_p,i} \wedge rel_{\mu_n,\mu_p}))$

5 **for** *every* rel_{μ_n,μ_p} **do**

6 EncodeDistRel$(\mu_n, lift_{\Pi}(\mu_p), R, rel_{\mu_n,\mu_p})$

7 AddCons$(\neg R_{s_N^0,1})$

Figure 5.5: SMT encoding for $N \not\preceq \mathcal{P}/\Pi$.

the quotient $\mathcal{P}/\Pi$ (line 7). A naïve encoding introduces a universal quantification over all possible strong simulations to say that no strong simulation relates the start states of a sample in $\mathcal{N}$ and $\mathcal{P}/\Pi$. We can avoid this by using the characterization of $\preceq$ in Lemma 14 for trees, as shown in Fig. 5.5. Here, we introduce Boolean variables $R_{s_n,i}$ to denote $s_n R e_i^{\Pi}$ for the coarsest strong simulation R between the tree N and $\mathcal{P}/\Pi$ and rel_{μ_n,μ_p} to denote $\mu_n \sqsubseteq_R lift_{\Pi}(\mu_p)$ for distributions $\mu_n \in Dist(S_N)$ and $\mu_p \in Dist(S_{\mathcal{P}})$. The constraints added on line 4 essentially encode the characterization of $\preceq$ in Lemma 14. In words, $s_n R e_i^{\Pi}$ holds iff for every transition $s_n \xrightarrow{a} \mu_n$, there exists some transition $s_p \xrightarrow{a} \mu_p$ in $\mathcal{P}$ on the same action a such that s_p belongs to the equivalence class e_i^{Π} and the lifting of μ_p is related to μ_n. The constraint on line 7 encodes that the coarsest strong simulation does not relate the start states of N and $\mathcal{P}/\Pi$. Finally, we encode the constraints on the variables rel_{μ_n,μ_p} as described in Chapter 4 (See Fig. 4.10). This needs us to encode the lifting $lift_{\Pi}(\mu_p)$ of a distribution μ_p to Π, which we do as follows.

Given $\mu_p \in Dist(S_{\mathcal{P}})$ and an equivalence class e_i^{Π}, $lift_{\Pi}(\mu_p)(e_i^{\Pi})$ can be encoded

as $\sum_{s \in Supp(\mu_p)} \ell_{\mu_p,i,s}$ where $\ell_{\mu_p,i,s}$ is a rational variable denoting the *contribution* of s towards the probability of the equivalence class e_i^{Π} in the lifted distribution, for which we add the constraints:

$$(\Pi_{s,i} \implies \ell_{\mu_p,i,s} = \mu_p(s)) \wedge (\neg\Pi_{s,i} \implies \ell_{\mu_p,i,s} = 0).$$

The following is immediate.

Lemma 22. *There exists a consistent partition of size k for samples $\mathcal{P}$ and $\mathcal{N}$ iff the constraints resulting from* EncodeConsisPartition$(\mathcal{P}, \mathcal{N}, k)$ *are satisfiable.*

5.2.2 Using Stochastic State-Space Partitioning

Consider again the positive and negative tree samples in Fig. 5.2. As mentioned in the previous subsection, H_1 in Fig. 5.3 is the smallest (w.r.t. the number of states) consistent LPTS that can be obtained using the state-space partitioning approach, but H_λ in the figure is also consistent with one less state. We can also show that there is no consistent LPTS with fewer states than H_λ – in order to simulate the positive sample, an LPTS with a single state should have self loops on all 3 actions which would then simulate all the negative samples as well. To be able to *fold* the positive sample P into H_λ, we need a way to group the states of P such that there is a one-to-one correspondence between the states and the transitions of the folding and H_λ. As we have seen with the above example, the state-space partitioning approach does not guarantee that and Lemma 21 shows an exponential upper bound on the number of states of the resulting folding. In this subsection, we will describe an

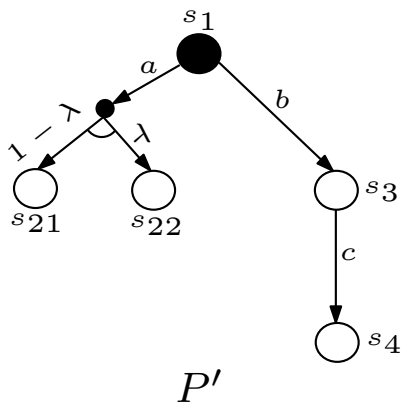


Figure 5.6: LPTS obtained by splitting s_2 of P in Fig. 5.2 into s_{21} and s_{22}.

alternative approach for folding the states to obtain consistent LPTSes of the least number of states.

We start with a high level description of the approach using the above mentioned example. Let R be a strong simulation between the positive sample P (Fig. 5.2) and H_λ (Fig. 5.3). Let the states of the LPTSes be labeled as shown in the figures. It is not hard to show that R must relate s_2 to both t_1 and t_2 in order for the transition on a in P to be simulated by H_λ. Now, consider the LPTS P' in Fig. 5.6 obtained from P by *splitting* s_2 into two states s_{21} and s_{22} such that the transition from s_1 on action a leads to s_{21} with probability $1-\lambda$ and s_{22} with probability λ. We can easily show that P is simulation equivalent to P' (i.e., $P \preceq P'$ and $P' \preceq P$) and so, we can consider P' as the positive sample, instead of P. But, more importantly, we can now obtain a partition $\Pi' = \{\{s_1, s_{21}\}, \{s_{22}, s_3, s_4\}\}$ of the state-space of P' whose quotient is exactly H_λ.

Alternatively, the above state splitting can be understood in the following way, in terms of the state-space of P. For each state of P, we assign a *probability distribution*

over a finite set, whose elements we call *groups*, as opposed to assigning a unique equivalence class in the case of a partition. For example, corresponding to the state splitting mentioned above, we have two groups, say g_1 and g_2, where s_2 is assigned the distribution which has probability $1-\lambda$ for g_1 and probability λ for g_2, s_1 is assigned the distribution δ_{g_1}, and s_3 and s_4 are assigned the distribution δ_{g_2}. In general, the distribution associated with a state (in other words, the splitting of a state) in the positive sample depends on the current group (in other words, the current split) of its parent. We formalize these ideas as a *stochastic partition* of $S_{\mathcal{P}}$, defined below. For $s \in S_{\mathcal{P}}$, we write $par(s)$ to denote the unique parent of s.

Definition 14 (Stochastic Partition)**.** *A* stochastic partition Π *of* $S_{\mathcal{P}}$ *is a tuple* $\langle G, g^0, D\rangle$ *where* G *is a finite set whose elements are called groups,* $g^0 \in G$*, and* $D : S \to (G \rightharpoonup \mathrm{Dist}(G))$*, such that the following hold. Let* $s \in S_{\mathcal{P}}$ *and* $g \in G$ *be arbitrary.*

1. *if* s *is a start state,* $D(s)(g) = \delta_{g^0}$*, and*
2. *if* s *is a not a start state,* $D(s)(g)$ *is defined iff* $g \in \mathrm{Supp}(D(par(s))(h))$ *for some* $h \in G$*.*

We write $[s]_\Pi$ *to denote the distribution map* $D(s)$*. When* Π *is clear from the context, we drop the subscript. For convenience, we write* $s \in g$ *to denote* $g \in \mathrm{Supp}(D(s)(h))$ *for some* $h \in G$ *and we sometimes confuse* g *with the set of all states* s *such that* $s \in g$*.*

Intuitively, a distribution assigned to a state s by a stochastic partition specifies how s and its incoming transitions are *split* which depends on how its parent was split. To see how to fold the positive samples, given a stochastic partition, we need

to define its *quotient*, analogous to the quotient of a partition. Note that, for a state s and a group g, $[s]_\Pi(g)$ is not always defined. For convenience, we extend $[s]_\Pi$ into a total function by letting $[s]_\Pi(g)(h) = 0$ for every group h, whenever $[s]_\Pi(g)$ is undefined.

Definition 15 (Quotient LPTS)**.** *Given a stochastic partition $\Pi = \langle G, g^0, D\rangle$ of $S_\mathcal{P}$, the* quotient *of Π, denoted $\mathcal{P}/\Pi$, is the LPTS $\langle G, g^0, \alpha, \tau\rangle$ where $\alpha = \bigcup_{P\in\mathcal{P}} \alpha_P$ and $(g, a, \mu) \in \tau$ iff there exists $(s, a, \mu_p) \in \tau_P$ for some $P \in \mathcal{P}$ with $s \in g$ and μ_p is lifted to Π to obtain μ as follows: for every $g' \in G$,*

$$\mu(g') = \sum_{s'\in g'} \left([s']_\Pi(g)(g') \cdot \mu_p(s')\right).$$

We write $lift_{\Pi,g}(\mu_p)$ to denote the lifting of μ_p to Π.

In other words, given $s \xrightarrow{a} \mu_p$, the lifting of μ_p assigns a probability to a group g' that is equal to the sum of the probabilities of all states under μ_p weighted by the probabilities of them being assigned to g' under Π. Moreover, this is in the context of the parent s being assigned to a group g. For instance, consider the transition $s_1 \xrightarrow{a} \delta_{s_2}$ of the positive sample P in the above example. Note that $g_1 = \{s_1, s_2\}$ and moreover, g_1 is the only group containing s_1. The probability of g_1 under the lifting of the distribution of this transition is obtained as

$$\begin{aligned}\sum_{s\in g_1} \left([s](g_1)(g_1) \cdot \delta_{s_2}(s)\right) &= [s_1](g_1)(g_1) \cdot \delta_{s_2}(s_1) + [s_2](g_1)(g_1) \cdot \delta_{s_2}(s_2)\\ &= 0 + (1-\lambda)\cdot 1\\ &= 1-\lambda\end{aligned}$$

Similarly, one can obtain the probability of g_2 under the lifting as λ. After lifting all the transitions of P in this way, one can see that the quotient is essentially H_λ where t_1 and t_2 in Fig. 5.3 correspond to g_1 and g_2, respectively.

Before moving on, we show that the quotient is a well-defined LPTS, i.e., the lifting of a distribution to a stochastic partition is a well-defined distribution over its groups.

Lemma 23. *Let Π be a stochastic partition of $S_\mathcal{P}$. Then, $\mathcal{P}/\Pi$ is a well-defined LPTS.*

Proof. Let G be the set of groups of Π and let $g \xrightarrow{a} \mu$ be a transition of $\mathcal{P}/\Pi$. It suffices to show that $\mu \in \mathit{Dist}(G)$. From Definition 15, there exists $s \xrightarrow{a} \mu_p$ for $s \in S_\mathcal{P}$ such that $s \in g$ and $\mu = \mathit{lift}_{\Pi,g}(\mu_p)$. Now,

$$
\begin{aligned}
\sum_{g' \in G} \mu(g') &= \sum_{g' \in G} \sum_{s' \in g'} ([s'](g)(g') \cdot \mu_p(s')) \\
&= \sum_{s' \in S_\mathcal{P}} \left(\mu_p(s') \cdot \sum_{\{g' \mid s' \in g'\}} [s'](g)(g') \right) \\
&= \sum_{s' \in S_\mathcal{P}} \left(\mu_p(s') \cdot \sum_{g' \in \mathit{Supp}([s'](g))} [s'](g)(g') \right) && \{\text{Definition 14}\} \\
&= \sum_{s' \in S_\mathcal{P}} \mu_p(s') && \{[s'](g) \in \mathit{Dist}(G)\} \\
&= 1 && \{\mu_p \in \mathit{Dist}(S_\mathcal{P})\}
\end{aligned}
$$

□

We have the following lemma analogous to state partitions.

Lemma 24. *Let* Π *be a stochastic partition of* $S_{\mathcal{P}}$*. Then,* $P \preceq \mathcal{P}/\Pi$ *for every* $P \in \mathcal{P}$*.*

Proof. Let $\Pi = \langle G, g^0, D \rangle$ and let $P = \langle S_P, s_P^0, \alpha_P, \tau_P \rangle$ be a sample in $\mathcal{P}$. Consider the relation $R = \{(s, g) \mid g \in G, s \in S_P \cap g\}$. By Definition 14, $s_P^0 R g^0$. To show $P \preceq \mathcal{P}/\Pi$, it suffices to show that R is a strong simulation.

Let sRg and $s \xrightarrow{a} \mu_p$. As $s \in g$, by Definition 15, $g \xrightarrow{a} \mu$ where $\mu = \mathit{lift}_{\Pi,g}(\mu_p)$. We will now show that $\mu_p \sqsubseteq_R \mu$. Let $S \subseteq \mathit{Supp}(\mu_p)$. We have

$$
\begin{aligned}
\mu_p(S) &= \sum_{s' \in S} \mu_p(s') \\
&= \sum_{s' \in S} \sum_{\{g' \mid s' \in g'\}} ([s'](g)(g') \cdot \mu_p(s')) && \{[s'](g) \in \mathit{Dist}(G)\} \\
&= \sum_{g' \in G} \sum_{s' \in S \cap g'} ([s'](g)(g') \cdot \mu_p(s')) \\
&= \sum_{g' \in R(S)} \sum_{s' \in S \cap g'} ([s'](g)(g') \cdot \mu_p(s')) && \{\text{Definition of } R\} \\
&\leq \sum_{g' \in R(S)} \sum_{s' \in g'} ([s'](g)(g') \cdot \mu_p(s')) \\
&= \sum_{g' \in R(S)} \mu(g') && \{\text{Definition 15}\} \\
&= \mu(R(S))
\end{aligned}
$$

So, by Lemma 11, $\mu_p \sqsubseteq_R \mu$. As $s \xrightarrow{a} \mu_p$ is arbitrary, we conclude that R is a strong simulation. □

We will now show that stochastic partitioning results in consistent LPTSes of the least number of states.

Lemma 25. *If L is an LPTS of k states that simulates all samples in $\mathcal{P}$, then there exists a stochastic partition Π of $S_{\mathcal{P}}$ of k groups with $\mathcal{P}/\Pi \preceq L$.*

Proof. Let $P \in \mathcal{P}$. Let $P = \langle S_P, s_P^0, \alpha_P, \tau_P \rangle$ and $L = \langle S_L, s_L^0, \alpha_L, \tau_L \rangle$. We know that $P \preceq L$. That is, there exists a strong simulation $R_P \subseteq S_P \times S_L$ with $s_P^0 R_P s_L^0$. Now, let $R = \bigcup_{P \in \mathcal{P}} R_P$. Let $s_p \in S_{\mathcal{P}}$ and $s_l \in S_L$. We assume a choice function *Witness* that given a pair $(s_p \xrightarrow{a} \mu_p, s_l)$ with $s_p R s_l$, outputs (μ_l, w) such that $s_l \xrightarrow{a} \mu_l$ and w is a weight function witnessing $\mu_p \sqsubseteq_R \mu_l$ according to Definition 8. Such a choice function always exists given that R is a strong simulation.

Let $s_p \in S_P$ for some $P \in \mathcal{P}$. We define the *depth* of s_p as its distance from the start state s_P^0. We define a stochastic partition $\Pi = \langle G, g^0, D \rangle$ where there is a one-to-one correspondence γ between S_L and G such that $\gamma(s_L^0) = g^0$ and D is defined as follows by induction on the depth of a state $s_p \in S_{\mathcal{P}}$. Using the same induction, we also show the following properties. Let $s_l \in S_L$ and $g \in G$.

1. $s_p \in \gamma(s_l)$ iff $s_p R s_l$ holds,
2. there exists an $h \in G$ such that $s_p \in h$,
3. if s_p is a start state, $D(s_p)(g) = \delta_{s_L^0}$, and
4. if s_p is not a start state, $D(s_p)(g)$ is defined iff $par(s_p) \in g$.

In the base case, s_p is a start state and we define $D(s_p)(g)$ to be the Dirac distribution $\delta_{s_L^0}$ for every $g \in G$. It is easy to see that the 4 properties mentioned above are satisfied for s_p.

In the inductive case, there is a unique transition $par(s_p) \xrightarrow{a} \mu_p$ such that $s_p \in Supp(\mu_p)$. Let $g \in G$ be arbitrary such that $par(s_p) \in g$ and let $g = \gamma(t_l)$. Such a group g is guaranteed to exist by inductive hypothesis. We now define $D(s_p)(g)$ for every such group g. Note also that, by inductive hypothesis, $par(s_p) R t_l$. Let $Witness(par(s_p) \xrightarrow{a} \mu_p, t_l) = (\mu_l, w)$. Then, we know that $t_l \xrightarrow{a} \mu_l$ and the weight function w witnesses $\mu_p \sqsubseteq_R \mu_l$. We then define $D(s_p)(g)$ to be a distribution $\mu \in Dist(G)$ such that $\mu(\gamma(s_l)) = w(s_p, s_l)/\mu_p(s_p)$ for every $s_l \in S_L$. It follows from Definition 8 that $\sum_{s_l \in S_L} \mu(\gamma(s_l)) = \sum_{s_l \in S_L} (w(s_p, s_l)/\mu_p(s_p)) = 1$ and hence, μ is well-defined. It is also easy to see that the 4 properties mentioned above are satisfied for s_p.

The 4 properties mentioned above immediately imply that Π is a well-defined stochastic partition, i.e., Π satisfies all the requirements of Definition 14.

We will now show that $\mathcal{P}/\Pi \preceq L$. Consider the binary relation $R' = \{(g, s_l) \mid g = \gamma(s_l)\}$. Clearly, $(g^0, s_L^0) \in R'$ by construction of Π. It suffices to show that R' is a strong simulation between $\mathcal{P}/\Pi$ and L.

Let $(g, s_l) \in R'$ and $g \xrightarrow{a} \mu$. Then, there exists $s_p \in g$ such that $s_p \xrightarrow{a} \mu_p$ and $\mu = lift_{\Pi,g}(\mu_p)$. By construction of Π, we know that $s_p R s_l$. Let $Witness(s_p \xrightarrow{a} \mu_p, s_l)$ output (μ_l, w) such that $s_l \xrightarrow{a} \mu_l$ and w is a weight function witnessing $\mu_p \sqsubseteq_R \mu_l$. We will show that $\mu \sqsubseteq_{R'} \mu_l$. Let $g' \in Supp(\mu)$ and $g' = \gamma(s'_l)$. We have,

$$\begin{aligned}
\mu(g') &= \sum_{s' \in g'} ([s'](g)(g') \cdot \mu_p(s')) \\
&= \sum_{s' \in \gamma(s'_l)} (D(s')(g)(\gamma(s'_l)) \cdot \mu_p(s')) && \{\text{notation}\}
\end{aligned}$$

$$= \sum_{s' \in \gamma(s'_l)} w(s', s'_l) \qquad \{\text{construction of } \Pi\}$$

$$= \sum_{s' R s'_l} w(s', s'_l) \qquad \{\text{property 1 above}\}$$

$$= \mu_l(s'_l) \qquad \{\text{Definition 8}\}$$

$$= \mu_l(R'(g'))$$

As g' is arbitrary, it follows from Lemma 11 that $\mu \sqsubseteq_{R'} \mu_l$. We conclude that R' is a strong simulation. □

The basic intuition behind the above lemma is that one can associate a group with each state in S_L and the weight/flow function that witnesses $\mu_p \sqsubseteq_R \mu_l$ for $\mu_p \in \mathit{Dist}(S_P)$ and $\mu_l \in \mathit{Dist}(S_L)$ identifies a splitting of the probabilities under μ_p to the states in $\mathit{Supp}(\mu_l) \subseteq S_L$. This splitting can then be used to define the distributions of a stochastic partition whose quotient is also consistent.

Our main result is immediate, using Lemmas 13 and 24. As in the case of partitions, we say that a stochastic partition Π is consistent iff $\mathcal{P}/\Pi$ is consistent.

Corollary 4. *For every consistent LPTS of k states, there is a consistent stochastic partition of k groups.*

Algorithm

As in the previous subsection, we can search for a consistent stochastic partition of the least size by iteratively checking if there exists a consistent stochastic partition of size k, for increasing values of k. Assuming that all probabilities involved

EncodeConsisStochPartition($\mathcal{P}, \mathcal{N}, k$)

1 introduce non-negative rational variables $\Pi_{s,i,j}$ to denote $[s]_\Pi(g_i^\Pi)(g_j^\Pi)$ for $(s,i,j) \in S_\mathcal{P} \times \{1,\ldots,k\} \times \{1,\ldots,k\}$

2 **for** $s \in S_\mathcal{P}$ *and* $1 \leq i \leq k$ **do**

    `// ` $[s]_\Pi(g_i)$ `is either well-defined or undefined`

3   AddCons($\left(\sum_{1\leq j\leq k} \Pi_{s,i,j} = 1\right) \vee \left(\sum_{1\leq j\leq k} \Pi_{s,i,j} = 0\right)$)

4 **for** $P \in \mathcal{P}$ *and* $1 \leq i \leq k$ **do**

    `// Condition 1 of Definition 14`

5   AddCons($\Pi_{s_P^0,i,1} = 1$)

6 **for** *every non-start state* $s \in S_\mathcal{P}$ *and* $1 \leq i \leq k$ **do**

    `// Condition 2 of Definition 14`

7   AddCons($\sum_{1\leq j\leq k} \Pi_{s,i,j} = 1 \iff \sum_{1\leq j\leq k} \Pi_{par(s),j,i} > 0$)

8 **for** $N \in \mathcal{N}$ **do**

    `// encode` $N \not\preceq \mathcal{P}/\Pi$

9   EncodeNotSim($N, \mathcal{P}, \Pi, k$)

Figure 5.7: SMT encoding for a consistent stochastic partition of size k for samples $\mathcal{P}$ and $\mathcal{N}$.

are rational, we can again encode each iteration as an SMT problem using EncodeConsisStochPartition($\mathcal{P}, \mathcal{N}, k$) in Fig. 5.7. Here, we introduce rational variables $\Pi_{s,i,j}$ to denote $[s]_\Pi(g_i^\Pi, g_j^\Pi)$ for some stochastic partition $\Pi = \langle\{g_1^\Pi, \ldots, g_k^\Pi\}, g_1^\Pi, D\rangle$. The constraints on lines 3, 5, and 7 essentially encode that the stochastic partition Π is well-defined.

Encoding consistency, i.e., that no sample in $\mathcal{N}$ is simulated by the quotient $\mathcal{P}/\Pi$, is similar to EncodeNotSim in Fig. 5.5 except that the equivalence classes are replaced by the groups of Π and $\Pi_{s,i}$ on line 4 is replaced by $\sum_{1\leq j\leq k} \Pi_{s,j,i} > 0$. Moreover, given $\mu_p \in \mathit{Dist}(S_\mathcal{P})$ and groups g_i^Π and g_j^Π, $\mathit{lift}_{\Pi,g_i^\Pi}(\mu_p)(g_j^\Pi)$ is encoded as $\sum_{s\in \mathit{Supp}(\mu_p)} (\Pi_{s,i,j} \cdot \mu_p(s))$.

The following is immediate.

Theorem 13. *There exists a consistent stochastic partition of size k for samples $\mathcal{P}$ and $\mathcal{N}$ iff the constraints resulting from* EncodeConsisStochPartition$(\mathcal{P}, \mathcal{N}, k)$ *are satisfiable.*

5.3 Convergence in Active Learning

Now that we have discussed algorithms for inferring an LPTS consistent with a given set of positive and negative samples (FindConsistent), we turn our attention to the convergence of the active learning framework LearnLPTS (see Fig. 5.1). As we have seen in Section 5.1, each iteration of LearnLPTS infers a consistent LPTS for the tree samples returned by the teacher so far with the ultimate goal of converging to an LPTS that is (simulation) equivalent to the unknown target. We start with a negative result which shows that under no assumptions about the samples the teacher can return, there is no converging solution to the learning problem.

Theorem 14. *There is no converging learning algorithm in the active learning framework* LearnLPTS.

Proof. Consider the LPTS U_λ in Fig. 5.9, parametric in a rational number $\lambda \in (0, 1)$. As shown in the figure, U_λ has one transition from the start state on a leading to a distribution μ_λ. Fig. 5.8 shows an adversarial teacher that manipulates the value of λ dynamically, as necessary, to ensure that a counterexample always exists no matter what LPTS the learner conjectures on line 4. Moreover, λ is updated in such a way that the new U_λ remains consistent with all the previously generated samples. This will ensure that the learner never converges. We describe the teacher below.

```
AdversarialTeacher()
1   λ ← arbitrary rational in (0, 1)
2   𝒩 ← ∅
3   while true do
4       H ← GetNextConjecture()
5       if H ⋠ U_λ then
6           obtain a tree counterexample N and add to 𝒩
7           return N as a negative sample
8       else if U_λ ⋠ H then
9           obtain a tree counterexample P
10          return P as a positive sample
11      else
12          λ⁺ = min({p_b^ν | p_b^ν > λ, ν ∈ Dist[a, 𝒩]} ∪ {1})
              // see text for description
13          λ ← (λ⁺ + λ)/2
14          obtain a tree counterexample P to U_λ ⪯ H
15          return P as a positive sample
```

Figure 5.8: An adversarial teacher in the proof of Theorem 14.

For every new conjecture made by the learner, the teacher first checks if there is a counterexample w.r.t. U_λ, for the current value of λ, and returns a positive or negative sample, as appropriate (lines 4–10). When the conjecture H results in no counterexamples, it increases the value of λ as follows. Let $Dist[a, \mathcal{N}] = \{\nu \mid s_N^0 \xrightarrow{a} \nu \text{ for some } N \in \mathcal{N}\}$ be the set of all distributions of the transitions on action a

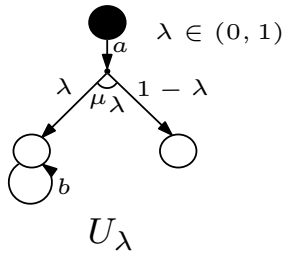


Figure 5.9: A target in the active learning framework where an adversarial teacher can dynamically modify the probability λ leading to the divergence of a learner.

outgoing from a start state in $\mathcal{N}$. Given $\nu \in Dist[a, \mathcal{N}]$, let p_b^ν be the cumulative probability under ν of all states in $Supp(\nu)$ that have at least one outgoing transition on action b. The idea is to consider the smallest of all such p_b^ν's that are larger than λ and to increase λ to a value in between, say the average (lines 12–13). If there is no p_b^ν that is greater than λ, we take the average between λ and 1. As λ is rational, this is always possible and the new value of λ remains in $(0, 1)$. It remains to show that U_λ, for the new value of λ, remains consistent with all the previously generated samples.

Let λ' be the new value of λ. By construction, $\lambda' > \lambda$ and hence, $U_\lambda \preceq U_{\lambda'}$. It follows from Lemma 13 that $U_{\lambda'}$ simulates every positive sample in $\mathcal{P}$.

Let N be a negative sample in $\mathcal{N}$. Assume, for the sake of contradiction, that $N \preceq U_{\lambda'}$. As the only outgoing transition from the start state of $U_{\lambda'}$ is on action a, every outgoing transition from the start state s_N^0 of N must also be labeled by a. Let $s_N^0 \xrightarrow{a} \nu$. Given our assumption that $N \preceq U_{\lambda'}$, we have that $\nu \sqsubseteq_\preceq \mu_{\lambda'}$. We can similarly reason that no state in $Supp(\nu)$ has a transition labeled by an action other than b. Moreover, every other transition in the tree N must also be labeled by b for $N \preceq U_{\lambda'}$ to hold. Consider p_b^ν, the cumulative probability under ν of all the states in $Supp(\nu)$ that have an outgoing transition on b. It follows from Lemma 11 that $p_b^\nu \leq \lambda'$. Given that N is a negative sample, $N \not\preceq U_\lambda$ and hence, $p_b^\nu > \lambda$. But then, from the above construction, $\lambda' < p_b^\nu$ holds which leads to a contradiction. We conclude that $N \not\preceq U_{\lambda'}$. □

At a high level, the above theorem holds because *it is not necessary* for the positive tree samples returned by the teacher to have an *execution mapping* to the

target U (see Section 4.5). As we have seen in the proof of the above theorem, this allows the possibility of adversarial behavior by deliberately choosing the probability values in the samples such that the learner is guaranteed not to converge. But, in practice, to be able to apply the learning framework in a given setting, we also need to implement the teacher and we are not aware of any algorithm to generate counterexamples other than the one discussed in Section 4.5. As mentioned before, this algorithm does have the property that the generated counterexample to $L_1 \preceq L_2$ has an *execution mapping* to L_1. This suggests us to impose the following *friendliness* condition on a teacher.

Condition 1 (Friendly Teacher)**.** *Every positive* (*negative*) *sample returned by the teacher should have an execution mapping to the target* (*conjecture*)*.*

First of all, note that the proof of the above theorem no longer works because updating the value of λ in Fig. 5.9 violates the above condition on previously returned positive samples. In fact, as we show below, using the state-space partitioning technique for inferring a consistent LPTS in each iteration (see Section 5.2.1) ensures convergence.

Lemma 26. *Under Condition 1 on the teacher, the learning algorithm that computes the quotient of a least-sized consistent state-space partition leads to convergence in the active learning framework* LearnLPTS*.*

Proof. Let U be the unknown target LPTS. Consider an arbitrary iteration of the learning loop in LearnLPTS. First of all, the execution mappings between the positive samples and U (which exist due to Condition 1) induce an equivalence relation among the states in $S_{\mathcal{P}}$ where two states are related iff they are mapped to

the same state of U. One can easily show that the quotient of the corresponding state-space partition is a *sub-structure* of U, i.e., U with some (if any) transitions removed, and hence, is trivially simulated by U. As U itself is a consistent LPTS, the quotient is also consistent (follows from Lemma 13). Therefore, the quotient of a consistent partition of the least size has at most $|S_U|$ number of states.

Now, one can show that there are only finitely many possible conjectures for a given number of states, across all iterations of LearnLPTS. This is because, from Condition 1, every distribution in $\mathcal{P}$ is a replica of some distribution of U, which are finitely many, and lifting a distribution to a partition only adds probabilities, which can only be done in finitely many ways. As every conjecture is inconsistent with the next sample, no two conjectures are simulation equivalent, and hence, no two conjectures are identical.

Together, we conclude that the learner converges. □

The following is immediate.

Theorem 15. *There exists a converging learning algorithm in the active learning framework under Condition 1 on the teacher.*

With our ultimate objective of deploying the learning algorithm for assume-guarantee compositional reasoning, it is desirable to learn an LPTS with as few states as possible. For this purpose, we will now impose the following condition on the learner to output only consistent LPTSes of the least number of states.

Condition 2 (Optimal Learner)**.** *Every conjecture H made by the learner is a consistent LPTS of the least number of states.*

However, there exists no converging learning algorithm under both Condition 1

and Condition 2, as shown below.

Theorem 16. *There is no converging learning algorithm in the active learning framework* LearnLPTS *under both Condition 1 on the teacher and Condition 2 on the learner.*

Proof. Let the LPTS H_1 in Fig. 5.3 be the unknown target. We will show that there is an adversarial strategy for the teacher to ensure that there is a counterexample for every conjecture made by the learner. In fact, we show that H_λ in Fig. 5.3, for a suitable value of λ, is a valid conjecture for every iteration of the learning loop for the adversarial strategy.

By Condition 2, the learner only conjectures least-state consistent LPTSes and so, the initial strategy of the teacher is to return samples until a 1-state LPTS H^* with self-loops on actions a, b, and c is conjectured. Until then, if a conjecture has transitions on an action other than a, b, and c, a negative sample with a single transition on that action is returned. Otherwise, the tree P in Fig. 5.3 is returned as the positive sample. Note that P has an execution mapping to H_1. It is easy to see that such samples can always be returned until the learner conjectures H^*.

Once H^* is conjectured, the teacher returns N_a in the figure to make sure that every future conjecture has at least 2 states. If a future conjecture has a transition on an action other than a, b, and c, the teacher can similarly return a negative sample as above. If not, it returns P or N_b in the figure, if possible. Otherwise, the teacher returns a new negative sample as follows.

Let s_1 and s_2 be the 2 states of the current conjecture H. Let Δ_a^i, Δ_b^i, and Δ_c^i be the sets of distributions of the transitions outgoing from s_i, $i \in \{1, 2\}$, on actions

a, b and c, respectively. As $N_a \not\preceq H$, we have that $\Delta_a^1 \neq \emptyset$ and for every $\mu_a \in \Delta_a^1$, $\mu_a(s_1) < 1$ and hence, $\mu_a(s_2) > 0$. Similarly, as $N_b \not\preceq H$ and $P \preceq H$, we have that $\Delta_b^1 \neq \emptyset$ and for every $\mu_b \in \Delta_b^1$, $\mu_b(s_2) > 0$ and every $s_i \in \mathit{Supp}(\mu_b)$, $\Delta_c^i \neq \emptyset$. That is, every transition on action b from s_1 has non-zero probability of going to s_2 and every state in its support has a transition on c. The teacher then returns $N_c^{\beta,\gamma}$ in the figure, where $\beta = \mu_a(s_2)$ for some $\mu_a \in \Delta_a^1$ and $\gamma = \mu_c(s_2)$ for some $\mu_c \in \Delta_c^2$. Note that $N_c^{\beta,\gamma}$ has an execution mapping to H.

It remains to show that there always exists a consistent LPTS of 2-states for the above adversarial strategy. We will show that H_λ in the figure is such a consistent LPTS. We have seen earlier in the chapter that H_λ simulates P and does not simulate either N_a or N_b. Moreover, if we choose $\lambda \in (0, \beta_{min})$, where β_{min} is the minimum value of β in all $N_c^{\beta,\gamma}$ samples returned so far, then H_λ does not simulate the current $N_c^{\beta,\gamma}$ either. Such a λ can always be chosen as $\beta_{min} > 0$.

We conclude that there is no converging learning algorithm under both Condition 1 on the teacher and Condition 2. □

Despite the above negative result, we obtain a semi-algorithm for the learning problem under both the conditions, by using *stochastic* state-space partitioning (Section 5.2.2) for FindConsistent in every iteration of the active learning framework. That is, if the learner converges, it is guaranteed to learn the target with the least number of states. Correctness is immediate from Theorem 13.

5.4 Learning Assumptions for Compositional Reasoning

We will now describe how the active learning framework LearnLPTS can be used for learning a sufficient intermediate assumption A in the rule Asym mentioned in Section 5.1. We start with the algorithms for the teacher and the learner and then briefly describe the complexity guarantees.

Teacher

The teacher simply performs two conformance checks corresponding to the two premises of the rule. If a conjecture A satisfies both the premises, the teacher returns *yes*. In this case, the conclusion holds as well, given that Asym is sound (Lemma 12). If one of the premises fails, the teacher generates a new sample with an *execution mapping*, using the counterexample generation algorithm described in Section 4.5. Thus, the teacher satisfies Condition 1. If premise 2 fails, a *positive* sample is returned to the learner. If premise 1 fails, the obtained counterexample C is first *projected* onto A, using the one-to-one correspondence from C to $L_1 \parallel A$ given by the execution mapping, and the projection is returned as a *negative* sample (see Section 6.3 for more details on projection).

Learner

The learner uses the state-space (stochastic) partitioning techniques described in Section 5.2 for inferring a new conjecture for the assumption A whenever a new

sample is returned by the teacher. As every positive sample has an execution mapping to L_2, the learning target is L_2 from the learner's perspective. This works because if the system $(L_1 \parallel L_2)$ conforms to P, then L_2 is clearly an assumption satisfying the premises. However, in practice, we expect the algorithm to converge to a smaller assumption that also satisfies the premises.

If the system conforms to the specification P, i.e., if the conclusion of ASym is actually true, then the learner is guaranteed to converge, provided it uses the state-space partitioning technique (Lemma 26). However, if it uses stochastic state-space partitions instead, convergence is not guaranteed as we saw in Section 5.3. Nevertheless, using stochastic partitions leads to a semi-algorithm for the problem of learning an intermediate assumption of the least-size.

If the system does not conform to P, however, there is no assumption satisfying both the premises of ASym (due to soundness of the rule). In this case, we are also interested in computing a counterexample to $L_1 \parallel L_2 \preceq P$. For this purpose, the learner performs a *spuriousness check* on the samples returned by the teacher, similar to the CEGAR approach [38]. We restrict the spuriousness check to *negative samples* following previous approaches [99]. In our case, the learner simply checks $N \preceq L_2$ for a negative sample N. If the check succeeds, then a counterexample can be constructed from the failure of $L_1 \parallel N \preceq P$. Otherwise, the learning framework moves on to the next iteration. A slightly more involved, but practical, way for detecting spuriousness of a negative sample is described in the next chapter.

Time Complexity Analysis

Let us now analyze the time complexity of assume-guarantee reasoning when state-space partitions are used by the learner. Note that, as described in Section 4.4, the time complexity of checking simulation conformance is polynomial in the sizes of the two LPTSes. So, the time complexity of monolithic reasoning to determine $L_1 \parallel L_2 \preceq P$ is $O(poly(|L_1| \cdot |L_2|, |P|))$, where $|L|$ denotes $\max(|S_L|, |\tau_L|)$, the maximum of the number of states and the number of transitions of L.

Let $d = |\tau_2|$ and b be the maximum size of the support of a distribution in L_2. Given a state of a candidate assumption of size k and a transition of L_2, there can be at most k^b-many corresponding outgoing transitions (taking non-determinism into account) from that state. For k states and d distributions, this gives an upper bound of dk^{b+1}. Therefore, there are $2^{dk^{b+1}}$ different possible candidates of size k to consider. If m is the number of states in the final assumption output by the algorithm, the total number of iterations of the learning algorithm is then given by $O(2^{dm^{b+1}})$. Note that $m = O(|S_2|)$, i.e., m is upper bounded by the number of states in L_2.

In each iteration, in the worst-case, the learning algorithm enumerates all the candidate assumptions of the current size k and performs simulation checks with all the negative samples. Each of these checks has a time complexity of $O(poly(|A|, |\mathcal{N}|, |N|_{max}))$, where A is the final assumption, $\mathcal{N}$ is the final set of negative samples and $|N|_{max}$ is the largest value of $|N|$, for any $N \in \mathcal{N}$. Thus, the total worst-case time complexity of the learning algorithm for computing the final assumption is $O(poly(|A|, |\mathcal{N}|, |N|_{max}) \cdot 2^{dm^{b+1}})$. Furthermore, the time complexity of checking the two premises of ASym (by the teacher) is $O(poly(|L_1| \cdot |A|, |P|) + poly(|L_2|, |A|))$ in

every iteration. We observe that, if the final assumption is small (i.e., $|A| \ll |L_2|$) in practice, this approach can be better than monolithic reasoning. Moreover, when A is small, we also expect memory savings in practice. In other cases, however, we would need better algorithms to address the problem.

5.5 Related Work

Learning for automating compositional reasoning of probabilistic systems has been proposed before [52] in the context of checking probabilistic reachability properties, which are refuted by sets of trace counterexamples. The approach uses a variant of L* [14], a learning algorithm for DFAs, to automatically learn deterministic assumptions, following previous work in the non-probabilistic setting [99]. The approach uses a sound but incomplete rule, and therefore, it is not guaranteed to terminate (completeness is necessary for termination). A complete rule for such properties restricted to systems without non-determinism has been considered recently [51]. It uses learning with *probabilistic* trace inclusion as the conformance relation which is undecidable. Also, the learning algorithm is not guaranteed to terminate. In contrast, we use simulation conformance which is decidable in polynomial time and leads to a sound and complete rule (Theorem 12). We are also able to guarantee termination for the algorithm proposed in Section 5.4 when using state-space partitions to infer a consistent LPTS.

Our work draws inspiration from a previous work [67] that automates assumption generation by using an algorithm for learning the *minimal separating automaton* from

positive and negative trace counterexamples. The counterexamples are provided via model checking in an assume-guarantee framework. Similar to our work, they use a *partitioning approach*, where the goal is to find a *folding* of the counterexamples into the learnt model. A different approach has been proposed to find the separating automaton based on L* which makes use of membership queries, in addition to equivalence queries [33]. All these works were done in the context of non-probabilistic reasoning under trace semantics and thus, are different from our setting.

Learning a minimum-state automaton from positive and negative samples is a well studied problem [15, 58, 97] that is known to be hard [61]. Algorithms have also been proposed for samples with stochastic information, i.e., the probability of acceptance of a trace or a tree [27, 28], learning stochastic finite (tree) automata. As also previously said, we cannot immediately borrow existing results from the above automata-theoretic approaches.

LPTSes are related to *probabilistic automata* (PA) [100]. Algorithms to learn PAs have only been proposed in restricted settings of stronger assumptions on a teacher [104] or approximate learning [44, 87]. Algorithms to learn a *multiplicity* automaton, which generalizes a PA by replacing the probabilities with arbitrary rationals, have also been proposed [21]. Adapting these to solve verification problems involving probabilistic transition systems is difficult and results in non-terminating algorithms [51]. On the other hand, we show in Section 5.4 that one can readily apply the algorithms we propose to infer intermediate assumptions in an automated assume-guarantee style framework for the verification of strong simulation conformance between LPTSes. This yields the first complete and fully automated learning

framework for compositional verification of probabilistic systems. Moreover, one can extend this framework to check logical properties, such as the fragment *weakly safe PCTL* [29], which are preserved by the conformance and also have tree counterexamples.

5.6 Conclusion

We have presented algorithms and decidability results for the problem of active learning for LPTSes from stochastic tree samples, using traditional and stochastic state-space partitioning. We have also described the application of the algorithms to automating the discovery of assumptions for the compositional verification of LPTSes.

In the future, it would be interesting to investigate further conditions on the teacher that will make the active learning problem with stochastic partitions decidable. The learning algorithms presented here are quite general and not restricted to compositional verification. So, another interesting future direction is to investigate new applications of our algorithms that may be in domains outside automatic verification.

The algorithms presented in this chapter are published as part of the proceedings of LICS 2012 [79].

Chapter 6

Abstraction Refinement for Simulation Conformance

6.1 Introduction

In previous chapters, we have described the problem of state-space explosion for checking simulation conformance of a multi-component *Labeled Probabilistic Transition System* (LPTS) against a specification LPTS. We have also described a framework for compositional reasoning in Chapter 5, using an *assume-guarantee paradigm*. To recall, the assume-guarantee reasoning we are interested in is captured by the following inference rule for the case of two components:

$$\frac{1 : L_1 \parallel A \preceq P \qquad 2 : L_2 \preceq A}{L_1 \parallel L_2 \preceq P} \text{ (ASYM)}$$

In the last chapter, we have seen iterative algorithms for *learning* a suitable intermediate assumption A that satisfies the premises of the above rule, utilizing counterexamples to the premises obtained from previous conjectures for A. In this chapter, we describe an alternative approach based on *automatic abstraction refinement* [38]. In this approach, the assumption A is maintained as a *conservative abstraction* of L_2, i.e., an LPTS that simulates L_2 (and hence, premise 2 holds by construction), and is iteratively refined based on tree counterexamples obtained from checking premise 1. Moreover, we use a state-space partitioning technique, similar to the one described in Section 5.2.1, for obtaining such an abstraction from L_2. This ensures that the iterative process is guaranteed to terminate, with the number of iterations bounded by the number of states of L_2.

We first describe an automatic abstraction refinement based algorithm for checking simulation conformance between two LPTSes in Section 6.2. This is based on the well-known *CounterExample Guided Abstraction Refinement* (CEGAR) approach for non-probabilistic systems [38]. We will then describe how to adapt CEGAR to the assume-guarantee setting, to obtain our algorithm *Assume-Guarantee Abstraction Refinement* (AGAR), in Section 6.3. When the system is composed of more than 2 components, i.e., when L_2 itself is composed of multiple components, we can apply assume-guarantee reasoning recursively for the second premise ($L_2 \preceq A$). We extend the rule ASym for the case of $n \geq 2$ components and describe how AGAR can be naturally extended to the new rule. We also briefly describe how AGAR can further be applied to the case where the required specification is given as a property in a logic preserved by strong simulation. We implemented the algorithms for counterex-

ample generation (described in Section 4.5) and for AGAR in Java using the Yices SMT solver [46] and show experimentally in Section 6.4 that AGAR can achieve significantly better performance than monolithic conformance checking.

Related Work. CEGAR algorithms for probabilistic systems have been proposed earlier in the context of probabilistic reachability [72] and safe-PCTL [29]. The CEGAR approach we describe in Section 6.2 is an adaptation of the latter. However, our main interest is in the compositional setting (Section 6.3).

6.2 CEGAR for Checking Strong Simulation

Assume that we are interested in checking whether $L \preceq P$ holds for 2 LPTSes $L = \langle S_L, s^0_L, \alpha_L, \tau_L \rangle$ and $P = \langle S_P, s^0_P, \alpha_P, \tau_P \rangle$. We describe an algorithm based on *automatic abstraction refinement* to infer an LPTS A that simulates L (in which case, we refer to A as an *abstraction* of L) while conforming to P, i.e., $L \preceq A \preceq P$. By Lemma 13, it will then follow that $L \preceq P$. The algorithm is based on the well-known *CounterExample Guided Abstraction Refinement* (CEGAR) approach [38]. Such a CEGAR approach can be useful when checking $L \preceq P$ directly is expensive. Moreover, we will see how CEGAR can be adapted to the assume-guarantee setting in Section 6.3.

Fig. 6.1 shows the pseudo-code of the CEGAR algorithm for simulation conformance between LPTSes. The algorithm maintains an abstraction A of L as the quotient of a state-space partition of L. The partition and the quotient construction are as described in Section 5.2.1. The algorithm initializes a variable Π, denoting a

```
CEGAR(L = ⟨S_L, s_L^0, α_L, τ_L⟩, P = ⟨S_P, s_P^0, α_P, τ_P⟩)
1   Π ← the coarsest partition of S_L
2   A ← L/Π
3   while true do
4       (res, C, M) ← CheckSim(A, P)
5       if res = yes then
6           return (yes, −)
7       (spurious, Π, R) ← AnalyzeAndRefine(C, L, Π, A, M)  // see text
8       if spurious then
9           A ← L/Π
10      else
11          return (no, C)
```

Figure 6.1: CEGAR algorithm for checking $L \preceq P$.

state-space partition of S_L, to the coarsest partition where there is only one equivalence class which contains all the states (line 1). The algorithm then iteratively refines Π (and hence, refines A) based on the counterexamples obtained from the simulation check $A = L/\Pi \preceq P$ (denoted by CheckSim on line 4). If $A \preceq P$ can be shown, we can conclude $L \preceq P$ (lines 5–6). Otherwise, we obtain a counterexample C to $L \preceq P$ which we need to check for feasibility in L (see below for details). For this purpose, we use the routine AnalyzeAndRefine (line 7), which we explain below. We also describe how the routine refines the partition Π in case C is *spurious*, which can then be used to update the abstraction A (line 9). However, if C is feasible in L, then we have found a counterexample to $L \preceq P$ which we return on line 11. Our counterexample analysis explained below is an adaptation of an existing one for counterexamples which are arbitrary *sub-structures* of A [29]; our stochastic tree counterexamples are not necessarily sub-structures of A.[1]

[1] A *sub-structure* of an LPTS L, at a high level, is an LPTS that can be obtained from L by removing some transitions. A formal presentation can be found elsewhere [29].

Spuriousness Check and Refinement (AnalyzeAndRefine). Let Π be a partition and $A = L/\Pi$ be such that $A \not\preceq P$ and let $C = \langle S_C, s^0_C, \alpha_C, \tau_C \rangle$ be a counterexample with an execution mapping M from S_C to S_A. Our goal is now to check whether C is feasible in L, i.e., whether $C \preceq L$. If it is feasible, C is a real counterexample as we already know $C \not\preceq P$. If it is not feasible, we need to refine A by refining the partition Π. Fig. 6.2 shows the pseudo-code for AnalyzeAndRefine, which is essentially an instrumented version of the algorithm for checking simulation conformance for trees from Section 4.4 (see ComputeSimTree in Fig. 4.11). With the goal of partition refinement, we compute the coarsest strong simulation between C and L contained in $R_M = \{(s_1, s_2) \mid s_1 \in S_C, s_2 \in S_L, M(s_1) = [s_2]_\Pi\}$, as opposed to $S_C \times S_L$ (line 1). In words, R_M relates a state s_1 of S_C to all states of S_L in the equivalence class $M(s_1)$. The analysis works as follows.

Intuitively, the algorithm does a bottom-up traversal of C and for each transition $s_1 \xrightarrow{a} \mu_1$, checks whether it is simulated by a transition from a state in $R(s_1)$, where R is the current value of the binary relation maintained by the algorithm. After every iteration of the for-loop beginning at line 4, the algorithm checks for the following two cases and refines the partition or continues with the next iteration. Let R_{old} be the value of R at the beginning of every iteration of this for-loop (see lines 2 and 20).

1. $R(s_1) = \emptyset$: There are two possible reasons for this case. One is that no state in $R_M(s_1)$ simulates all the outgoing transitions of s_1, in which case the equivalence class $M(s_1)$ is a candidate for refinement. The other reason is that R does not relate the states in $Supp(\mu_1)$ sufficiently enough to the states in L for the transition to be simulated by a state in $R_M(s_1)$. In this case,

the equivalence classes corresponding to the states in $Supp(\mu_1)$ are candidates for refinement. As shown on lines 14–15, we refine every equivalence class in $\{M(s) \mid s \in \{s_1\} \cup Supp(\mu_1)\}$ by splitting $M(s)$ into $R_{old}(s)$, which is a subset of $M(s)$ by construction, and the rest.

2. $R(s_1) \neq \emptyset$, but $M(s_1) = [s_L^0]_\Pi$ and $s_L^0 \in R_{old}(s_1) \setminus R(s_1)$: In this case, $M(s_1)$ is the initial state of the abstraction A but s_1 is no longer related to the initial state s_L^0 of L. Here, the equivalence class $M(s_1)$ is a candidate for refinement and we split it into $R_{old}(s_1) \setminus R(s_1)$ and the rest (line 18).

The following lemma shows that the above mentioned refinement strategy is guaranteed to result in a *strictly finer* partition and hence, refine the abstraction A.

Lemma 27. *If* AnalyzeAndRefine(C, L, Π, A, M) *returns* $(yes, \Pi', -)$, *then* Π' *is a strictly finer partition than* Π, *i.e.,* $\Pi' < \Pi$.

Proof. It suffices to show that on lines 15 and 18, at least one equivalence class gets split into two non-empty subsets.

Consider the first case where $R(s_1) = \emptyset$ for some $s_1 \in S_C$ on line 13 of the pseudo-code in Fig. 6.2. If $R_{old}(s_1) \subsetneq R_M(s_1)$, then we are done. Otherwise, it must be the case that $R_{old}(s) \subsetneq R_M(s)$ for some $s \in Supp(\mu_1)$. This is because, s_1 and μ_1 are related to A by the execution mapping M and the corresponding distribution in A is obtained by lifting a distribution of L. So, if $R_{old}(s) = R_M(s)$ for every $s \in Supp(\mu_1)$, one can then show that $R(s_1)$ could not have been empty.

In the second case, $s_L^0 \in R_{old}(s_1) \setminus R(s_1)$ and $\emptyset \neq R(s_1) \subseteq R_{old}(s_1)$. Clearly, splitting $M(s_1)$ results in two non-empty subsets. □

```
AnalyzeAndRefine(C, L, Π, A = L/Π, M : S_C → S_A)
    // BEGIN INSTRUMENTATION
1   R ← {(s_1, s_2) | s_1 ∈ S_C, s_2 ∈ S_L, M(s_1) = [s_2]_Π}
    // relate s_1 with all states in the equivalence class M(s_1)
2   R_old ← R
    // END INSTRUMENTATION

3   for every non-leaf s_1 ∈ S_C in a bottom-up traversal of C do
4       for every s_1 -a-> μ_1 do
5           for every s_2 ∈ R(s_1) do
6               sim ← false
7               for every s_2 -a-> μ_2 do
8                   if μ_1 ⊑_R μ_2 then
9                       sim ← true
10                      break
11              if sim = false then
12                  R ← R \ {(s_1, s_2)}

                // BEGIN INSTRUMENTATION
13          if R(s_1) = ∅ then
14              for every s ∈ {s_1} ∪ Supp(μ_1) do
15                  refine Π by splitting M(s) into R_old(s) and the rest
16              return (yes, Π, −)
17          else if M(s_1) = [s_L^0]_Π and s_L^0 ∉ R(s_1) then
18              refine Π by splitting M(s_1) into R_old(s_1) \ R(s_1) and the rest
19              return (yes, Π, −)
20          R_old ← R
            // END INSTRUMENTATION
21  return (no, −, R)
```

Figure 6.2: Counterexample analysis and partition refinement, obtained by instrumenting ComputeSimTree in Fig. 4.11.

If neither of the above mentioned cases for refinement is encountered before reaching the end (line 21), clearly, $R(s_C^0) \neq \emptyset$ and $s_L^0 \in R(s_C^0)$ and hence, the final value of R is a strong simulation between C and L that relates the initial states. In other words, C is a real counterexample.

6.3 Assume-Guarantee Abstraction Refinement

We will now describe how CEGAR can be adapted to the assume-guarantee setting, which we call *Assume-Guarantee Abstraction Refinement* (AGAR). The notable difference with CEGAR is that the counterexample analysis is performed in an assume guarantee style: a counterexample obtained from checking one component (together with an abstraction of the environment, i.e., the other components) is used to refine the abstraction of a different component.

Fig. 6.3 shows the pseudo-code of our AGAR algorithm. Given LPTSes L_1, L_2 and P, the goal is to check $L_1 \parallel L_2 \preceq P$ in an assume-guarantee style, using the rule ASym. The basic idea is to maintain A in the rule as an abstraction of L_2, i.e., the second premise ($L_2 \preceq A$) holds for free throughout, and to check only the first premise ($L_1 \parallel A \preceq P$) for the abstraction A maintained by the algorithm. As in CEGAR, we restrict A to the quotient of a state-space partition of S_2, the states of L_2. If the first premise holds for A, then $L_1 \parallel L_2 \preceq P$ also holds, by the soundness of the rule. Otherwise, the obtained counterexample C is analyzed to see whether it indicates a real error or is spurious, in which case A is refined. The spuriousness analysis and refinement are compositional, as explained below.

```
AGAR(L_1, L_2, P)
    Π ← the coarsest partition of S_2
    A ← L_2/Π
    while true do
        (res, C, M) ← CheckSim(L_1 ∥ A, P)
        if res = yes then
            return (yes, −)
        (C⌊_A, M⌊_A) ← Project(C, A, M)
        (spurious, Π, R) ← AnalyzeAndRefine(C⌊_A, L_2, Π, A, M⌊_A)
        if spurious then
            A ← L_2/Π
        else
            return (no, C)
```

Figure 6.3: AGAR algorithm for checking $L_1 \parallel L_2 \preceq P$ in an assume-guarantee style.

Analysis and Refinement. In AGAR, the counterexample analysis is performed compositionally, using the *projections* of C onto L_1 and A obtained as follows. As in the case of CEGAR, we make use of an execution mapping M from S_C to $S_{L_1 \parallel A}$. From Definition 12, we know that every state of $L_1 \parallel A$ is a pair of states of L_1 and A, and every distribution of a transition in $L_1 \parallel A$ is the product of two distributions, one each from L_1 and A. So, we can utilize the one-to-one correspondence from the states and distributions of C to those of $L_1 \parallel A$, given by M, and pick the respective components of the state pairs and products of distributions to obtain what we call *projections* of C onto L_1 and A. We denote these projections by $C\lfloor_{L_1}$ and $C\lfloor_A$, respectively. Note that there is a natural *execution mapping* from $C\lfloor_A$ to A, which we denote by $M\lfloor_A$ (line 5). We will then use AnalyzeAndRefine, described in Section 6.2, to check whether $C\lfloor_A$ is simulated by L_2 or if it is spurious and refine the partition and the abstraction A (line 6). If AnalyzeAndRefine returns *no*, i.e., it concludes that $C\lfloor_A$ is not spurious, we can conclude that C is a counterexample to

$L_1 \parallel L_2 \preceq P$, as the following lemma shows.

Lemma 28. *If* AnalyzeAndRefine *returns* no *at line 6 of* Agar*, then then C, found on line 4, is a counterexample to $L_1 \parallel L_2 \preceq P$.*

Proof. For an LPTS $L = \langle S_L, s_L^0, \alpha_L, \tau_L \rangle$ and alphabet β such that $\alpha_L \subseteq \beta$, we write L^β to denote the LPTS $\langle S_L, s_L^0, \beta, \tau_L \rangle$. Let S_i and α_i be the set of states and alphabet for the LPTS L_i.

If AnalyzeAndRefine returns *no*, then $C{\downarrow_A} \preceq A$ holds. So, $(C{\downarrow_A})^{\alpha_2} \preceq L_2$ also holds. Moreover, we know that the projection $C{\downarrow_{L_1}}$ satisfies $(C{\downarrow_{L_1}})^{\alpha_1} \preceq L_1$. Together, it follows that $(C{\downarrow_{L_1}})^{\alpha_1} \parallel (C{\downarrow_A})^{\alpha_2} \preceq L_1 \parallel L_2$ (Lemmas 19 and 13). Moreover, the projections can also be shown to satisfy $C \preceq (C{\downarrow_{L_1}})^{\alpha_1} \parallel (C{\downarrow_A})^{\alpha_2}$. As $C \not\preceq P$ is already known, it follows from Lemma 13 that C is a *real* counterexample. □

The following is immediate from Theorem 12 and Lemmas 27 and 28.

Theorem 17 (Correctness and Termination)**.** Agar *is guaranteed to terminate within $|S_2| - 1$ iterations, the conformance $L_1 \parallel L_2 \preceq P$ holds iff it returns on line 11 and fails to hold iff it returns on line 10.*

In practice, we expect AGAR to take less than $|S_2| - 1$ iterations, terminating with an assumption smaller than L_2. AGAR will terminate as soon as it finds an assumption that satisfies the premises or that helps exhibit a real counterexample. Note also that, although AGAR uses an explicit representation for the individual components, it never builds $L_1 \parallel L_2$ directly (except in the worst-case) keeping the cost of verification low.

For example, Fig. 6.4 shows a specification LPTS for the 2-component input-

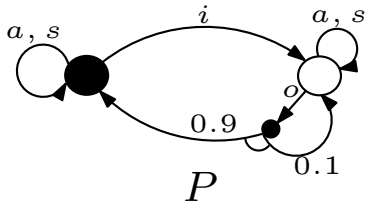


Figure 6.4: A specification for $L_1 \parallel L_2$, where L_1 and L_2 are in Fig. 4.16.

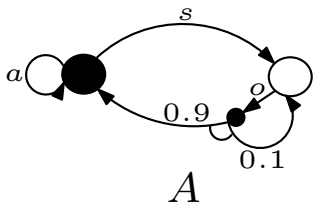


Figure 6.5: An assumption for $L_1 \parallel L_2$ in Fig. 4.16 and specification P in Fig. 6.4.

output system we saw in Fig. 4.16. For this example, our algorithm AGAR generates the sufficient assumption A shown in Fig. 6.5.

6.3.1 Reasoning with $n \geq 2$ Components

So far, we have discussed assume-guarantee reasoning in the context of two components L_1 and L_2. This reasoning can be generalized to $n \geq 2$ components using the following rule ASYM-N (which can similarly be shown to be sound and complete). This rule enables us to overcome the *intermediate state-space explosion* that may be associated with two-way decompositions when the subsystems are larger than the entire system.

$$\frac{1 : L_1 \parallel A_1 \preceq P \qquad 2 : L_2 \parallel A_2 \preceq A_1 \qquad \ldots \qquad n : L_n \preceq A_{n-1}}{\|_{i=1}^{n} L_i \preceq P} \text{ (ASYM-N)}$$

```
AGAR-N(⟨L_1, ..., L_n⟩, A_n, P)
1   L ← L_n
2   if A_n ≠ ϵ then // A_n = ϵ holds only for the very first call
3       L ← L ∥ A_n
4   if n = 1 then
5       (res, C, M) ← CheckSim(L, P)
6       (C⌊_{A_n}, M⌊_{A_n}) ← Project(C, A_n, M)
7       return (res, C, M)
    // compute sufficient assumption A for (L_1 ∥ ··· ∥ L_{n−1}) ∥ L ⪯ P
8   Π ← coarsest partition of S_L
9   A ← L/Π
10  while true do
11      (res, C, M) ← AGAR-N(⟨L_1, ..., L_{n−1}⟩, A, P)
12      if res = yes then
13          return (yes, −, −)
        // C ⪯ A with execution mapping M
14      (spurious, Π, R) ← AnalyzeAndRefine(C, L, Π, A, M)
15      if spurious then
16          A ← L/Π
17      else
            // R is a strong simulation between C and L
            // but we also need an execution mapping
18          (T_L, M_L) ← ObtainCexWithMapping(C, L, R)
19          (T_L⌊_{A_n}, M_L⌊_{A_n}) ← Project(T_L, A_n, M_L)
20          return (no, T_L⌊_{A_n}, M_L⌊_{A_n})
```

Figure 6.6: AGAR algorithm for checking $L_1 \parallel \cdots \parallel L_n \parallel A_n \preceq P$ for $n \geq 2$.

Fig. 6.6 shows the pseudo-code for AGAR-N, the adaptation of AGAR for the rule ASym-N. The algorithm takes as input n components $\langle L_1, \ldots, L_n \rangle$ and an *environmental assumption* A_n, which abstracts the rest of the components (if any), which should together conform to a specification P. Initially, the environmental assumption is absent, which we denote by letting A_n to be ϵ. AGAR-N then tries to compute an assumption A, in the sense of the two-component rule ASym, by using the two-way decomposition $(L_1 \parallel \cdots \parallel L_{n-1}) \parallel (L_n \parallel A_n) \preceq P$ as follows.

It first composes L_n with A_n, to obtain an LPTS L (lines 1–3). If there was only one component to begin with (line 4), it performs a monolithic simulation conformance check (denoted by CheckSim on line 5) and returns the result. If a counterexample is returned on line 5, along with an execution mapping M, it is projected onto the assumption A_n (line 6) so that the caller can check for spuriousness and refine A_n. On the other hand, if $n > 1$, the desired assumption A is computed by means of abstraction refinement of L (lines 8–20), similar to AGAR, recursively invoking AGAR-N for $\langle L_1, \ldots, L_{n-1} \rangle$ with the current abstraction A of L as the environmental assumption (line 11).

If the recursive call to AGAR-N returns *yes*, then A is a sufficient assumption (line 13). Otherwise, we obtain a counterexample C with an execution mapping M from S_C to S_A. The goal is now to check whether C is spurious and refine A in case it is. This is done similar to AGAR (line 14–16). If C is not spurious, then the environmental assumption A_n needs to be checked for refinement. But, we cannot simply project C onto A_n, as C does not have an execution mapping to L. However, the call to AnalyzeAndRefine on line 14 also returns a strong simulation R between C and L that relates the initial states. We can then use R to obtain a new tree T_L by simulating every transition in C, say in a top-down traversal of C (ObtainCexWithMapping on line 18). We can show that $C \preceq T_L \preceq L$ and hence, we can use T_L instead of C. As T_L is essentially an unrolling of L, we can also obtain an execution mapping to L, say M_L. We can then project T_L and M_L onto A_n (line 19) which are returned to the caller (line 20).

6.3.2 Compositional Verification of Logical Properties

AGAR can be further applied to automate assume-guarantee verification of properties written as formulae in a logic that is preserved by strong simulation, such as the *weak-safety* fragment of probabilistic CTL (PCTL) [29] which also admits tree counterexamples. The modified rule ASYM is both sound and complete for this logic ($\models$ denotes property satisfaction), provided $\alpha_A \subseteq \alpha_2$ with a proof similar to that of Theorem 12.

$$\frac{1 : L_1 \parallel A \models \phi \qquad 2 : L_2 \preceq A}{L_1 \parallel L_2 \models \phi}$$

The intermediate assumption A can be similarly maintained as a conservative abstraction of L_2 and iteratively refined based on the tree counterexamples to premise 1, using the same procedures as before. The rule can be generalized to reasoning about $n \geq 2$ components as described above and also to richer logics with more general counterexamples adapting existing CEGAR approaches [29] to AGAR. We plan to further investigate this direction in the future.

6.4 Implementation and Results

Implementation. We implemented the algorithms for checking strong simulation and generating counterexamples, which we described in Chapter 4, as well as AGAR and AGAR-N in Java. We used the front-end of the tool PRISM [82] to parse

the models of the components described in PRISM's input language and construct LPTSes which are then handled by our implementation.

As mentioned in Chapter 4, in order to take advantage of the efficient SMT solvers that exist today, we used the SMT encoding given by EncodeSim in Fig. 4.9 for checking simulation conformance. It follows from Lemma 15 that the constraints generated by the SMT encoding are satisfiable iff the conformance check succeeds. However, when the conformance check fails to hold, there is no direct way to obtain a tree counterexample. Note that in this case, the constraints generated by the SMT encoding are unsatisfiable. So, when the conformance check fails to hold, say between LPTSes L_1 and L_2, we obtain an unsatisfiable subset of the constraints, by utilizing the *unsat core extraction* facility provided by the Yices SMT solver. From this subset of constraints, we then construct a *sub-structure* of L_1 and check the conformance of this sub-structure against L_2 using the Java implementation. This sub-structure is usually much smaller than L_1 and contains only the information necessary to expose the counterexample.

Results. We evaluated our algorithms using this implementation on several examples analyzed in previous work [52]. Some of these examples were created by introducing probabilistic failures into non-probabilistic models used earlier [99] while others were adapted from PRISM benchmarks [82]. The properties used previously were about *probabilistic reachability* and we created our own specification LPTSes after developing an understanding of the models. The models in all the examples satisfy the respective specifications. We briefly describe the models and the specifications

below.[2]

CS_1 and CS_N model a *Client-Server* protocol with mutual exclusion having probabilistic failures in one or all of the N clients, respectively. The specifications describe the probabilistic failure behavior of the clients while hiding some of the actions as is typical in a high level design specification.

MER models a *resource arbiter* module of NASA's software for *Mars Exploration Rovers* which grants and rescinds shared resources for several users. We considered the case of two resources with varying number of users and probabilistic failures introduced in all the components. As in the above example, the specifications describe the probabilistic failure behavior of the users while hiding some of the actions.

SN models a wireless *Sensor Network* of one or more sensors sending data and messages to a process via a channel with a bounded buffer having probabilistic behavior in the components. Creating specification LPTSes for this example turned out to be more difficult than the above examples, and we obtained them by observing the system's runs and by manual abstraction.

Table 6.1 shows the comparison of running time and memory consumption among ASym, ASym-N, and monolithic (non-compositional) conformance checking. *Time* is in seconds and *Memory* is in megabytes. Table 6.2 compares the sizes of various LPTSes constructed by the approaches. $|X|$ stands for the number of states of an LPTS X. L stands for the whole system, P for the specification, L_M for the LPTS

[2]All models and specifications are available at
http://www.cs.cmu.edu/~akomurav/projects/agar/AGAR.html.

Example (*param*)	ASym		ASym-N		Mono	
	Time	*Mem*	*Time*	*Mem*	*Time*	*Mem*
$CS_1(5)$	7.2	15.6	74.0	15.1	**0.2**	**8.8**
$CS_1(6)$	11.6	22.7	810.7	21.4	**0.5**	**12.2**
$CS_1(7)$	37.7	49.4	*out*	–	**0.8**	**17.9**
$CS_N(2)$	0.7	7.1	2.4	6.8	**0.1**	**5.9**
$CS_N(3)$	43.0	63.0	1.6k	109.6	**14.8**	**37.9**
$CS_N(4)$	*out*	–	*out*	–	**1.8**k	**667.5**
MER (3)	**2.6**	19.7	3.6	**14.6**	193.8	458.5
MER (4)	**15.0**	53.9	34.7	**37.8**	*out*	–
MER (5)	–	*out*[1]	**257.8**	**65.5**	–	*out*[1]
SN (1)	**0.2**	**6.2**	1.7	8.5	1.5	27.7
SN (2)	**79.5**	**112.9**	694.4	171.7	4.7k	1.3k
SN (3)	*out*	–	**7.2**k	**528.8**	–	*out*

Table 6.1: Time and Memory consumption for AGAR vs monolithic verification. [1] Mem-out during model construction.

Example (*param*)	ASym				ASym-N			Mono	
	$\|L_1\|$	$\|L_2\|$	$\|L_M\|$	$\|A_M\|$	$\|L_c\|$	$\|L_M\|$	$\|A_M\|$	$\|L\|$	$\|P\|$
$CS_1(5)$	36	405	182	33	36	182	34	**94**	16
$CS_1(6)$	49	1215	324	41	49	324	40	**136**	19
$CS_1(7)$	64	3645	538	56	64	–	–	**186**	22
$CS_N(2)$	25	9	51	7	9	40	25	**34**	15
$CS_N(3)$	125	16	324	12	16	372	125	**184**	54
$CS_N(4)$	625	25	–	–	25	–	–	**960**	189
MER (3)	278	1728	**706**	7	278	**706**	7	16k	12
MER (4)	465	21k	**2**k	11	465	**2**k	11	120k	15
MER (5)	700	250k	–	–	700	**3.3**k	16	841k	18
SN (1)	43	32	**43**	3	126	165	6	462	18
SN (2)	796	32	**796**	3	252	1.4k	21	7860	54
SN (3)	7545	32	–	–	378	**1.4**k	21	78k	162

Table 6.2: Sizes of various LPTSes constructed for AGAR vs monolithic verification. [1] Mem-out during model construction.

with the largest number of states built by composing LPTSes during the course of AGAR, A_M for the assumption with the largest number of states during the execution and L_c for the component with the largest number of states in ASym-N. We also compared $|L_M|$ with $|L|$, as $|L_M|$ denotes the largest LPTS ever built by AGAR. Best figures, among ASym, ASym-N and Mono, for *Time*, *Memory* and LPTS sizes, are boldfaced. All the results were obtained on a Fedora-10 64-bit machine running on an Intel® Core™2 Quad CPU of 2.83GHz and 4GB RAM. We imposed a 2GB upper bound on Java heap memory and a 2 hour upper bound on the running time.

We observed that most of the time during AGAR was spent in checking the premises and an insignificant amount was spent for the composition and the refinement steps. Also, most of the memory was consumed by Yices. We tried several orderings of the components (the L_i's in the rules) and report only the ones giving the best results.

While monolithic checking outperformed AGAR for *Client-Server*, there are significant time and memory savings for *MER* and *Sensor Network* where in some cases the monolithic approach ran out of resources (time or memory). One possible reason for AGAR performing worse for *Client-Server* is that $|L|$ is much smaller than $|L_1|$ or $|L_2|$. When compared to using ASym, ASym-N brings further memory savings in the case of *MER* and also time savings for *Sensor Network* with parameter 3 which could not finish in 2 hours when used with ASym. As already mentioned, these models were analyzed previously with an assume-guarantee framework using learning from traces [52]. Although that approach uses a similar assume-guarantee rule (but instantiated to check *probabilistic reachability*) and the results have some similarity (e.g. *Client-Server* is similarly not handled well by the compositional approach), we can not directly compare it with AGAR as it considers a different class of properties.

6.5 Conclusion

We described a complete, fully automated abstraction-refinement approach for assume-guarantee reasoning of strong simulation conformance between LPTSes. The approach uses refinement based on stochastic tree counterexamples and it further ap-

plies to checking *safe*-PCTL properties. We showed experimentally the merits of the proposed technique in comparison to monolithic conformance checking. In future, it would be interesting to extend the approach to cases where the assumption A is allowed to have a smaller alphabet than that of the component it abstracts as this can potentially lead to further savings. Strong simulation would no longer work and one would need to use *weak* simulation [102], whose decidability is not known yet to the best of our knowledge. In an orthogonal direction, it is interesting to explore symbolic implementations of our algorithms, for increased scalability. An experimental comparison of the approach presented in this chapter with the active learning based algorithms from the previous chapter is also interesting. However, this requires one to first evaluate the algorithms from the previous chapter from a practical perspective and/or investigate practical implementations of the algorithms.

The results presented in this chapter are published as part of the proceedings of CAV 2012 [78].

Chapter 7

Future Work

There are several exciting research directions one can extend the thesis work to in the future. We will briefly describe some of them below.

Proof Based Abstraction from different under-approximations. As we have mentioned right in Chapter 1, a central theme of SAT/SMT-based model checking is to iteratively solve *Bounded Model Checking* (BMC) problems, obtain proofs of bounded safety, and try to generalize the proofs to invariants of the entire program. More generally, the strategy used is to under-approximate and generalize, in an iterative manner. In that sense, the approach presented in Chapter 3 adds an important component to the generalization step by means of *Proof Based Abstraction*. It is interesting to note that the approach, as described in Fig. 3.2, is quite general and not restricted to bounding the number of iterations of a loop. For example, one can obtain different kinds of under-approximations by bounding the range sets of the program variables, or the stack-depth in a recursive program, or the number of

context-switches in a concurrent program. It would be particularly interesting to explore these ideas for concurrent programs as such programs are notoriously difficult to verify.

Syntactic vs. Semantic Abstractions. We have seen several abstraction and approximation mechanisms in the algorithms described in Chapters 2 and 3. In the latter, especially in our implementation described in Section 3.5, we use a combination of a constraint-based method and program invariants to obtain an abstraction. While a constraint-based method depends on the input *syntactic* structure of the transition relation, an invariant is primarily a *semantic* artifact – a formula that over-approximates the reachable set of states. On the other hand, the abstraction mechanism described in Chapter 2 is purely semantic – we use formulas over the input-output parameters of a procedure to approximate its behavior. It would be interesting to explore a combination of the two abstraction mechanisms.

Synthesizing Ghost-code for Verification. In the literature on deductive verification, the term *ghost-code* is used to refer to a piece of code that is added specifically to assist in verification and which does not interfere with the execution of the original program (I do not know the origin of the term, but see Filliâtre et al. [53] for a recent reference). However, to the best of our knowledge, there is no automatic verification method that has a seamless integration with synthesizing and utilizing appropriate ghost-code. The program transformation we describe in Section 3.5 adds code to count the number of iterations of a loop which is one of the simplest kinds of ghost-code. It would be interesting to explore the use of other, more expressive, kinds of ghost-code, e.g., synthesizing useful terms or predicates over program vari-

ables (inspired by Predicate Abstraction [62]) and treating them as first-class objects as opposed to artifacts in an external verification mechanism.

Verification of Evolving Software. Real software is constantly evolving with design modifications, addition of new functionality, improvements in efficiency, etc. An exhaustive verification of the entire software after every change can be very expensive. It is interesting to consider how we can reuse the verification efforts from a previous version of the software for verifying the current version. For example, if we have an abstraction of the previous version of the software that continues to be an abstraction of the current version, then we can simply reuse the invariants computed in the past. However, this is only the best case scenario and there have been some recent attempts at localizing verification efforts by making use of previous abstractions (e.g., [50]). In general, we also need effective ways of *translating* the artifacts, such as invariants, from previous versions to maximize reuse and minimize verification efforts.

Handling Richer Logical Theories. The algorithms we described in this dissertation can handle programs over any decidable logical theory. However, the approximation techniques for existential quantification described in Chapter 2, which are essential to avoid the exponentially growing BMC problems, are restricted to Linear Rational Arithmetic and Presburger Arithmetic. In particular, if the theory has uninterpreted functions, it is not possible, in general, to eliminate existential quantifiers given the undecidability of first-order logic. This raises the important question of how we can efficiently approximate existential quantification in order to have a useful compositional verification approach. Handling theories with non-linear

real functions is another very important problem given the rise of cyber-physical systems. There have been recent advancements in efficient and practical constraint solving for such theories [56] and Bounded Model Checking using satisfiability over such theories [57], but extending the techniques to scalable *unbounded* verification remains an open challenge.

Verifying programs that manipulate pointers and arrays is another important problem which poses significant challenges given that even the simplest of the programs require universal quantifiers in the invariants. Given the rise of scalable verification algorithms for inferring quantifier-free invariants (including the algorithms developed in this dissertation), one approach is to iteratively reduce to the problem of inferring quantifier-free invariants [23]. This involves finite heuristic instantiation of the universal quantifiers with ground terms and has been shown to work successfully for small textbook examples in the above reference. However, scaling such an approach to handle realistic programs remains a challenge.

Efficient Probabilistic Analysis. With the growing complexity of software, a practical alternative for scaling verification technology is to focus on *probabilistic correctness guarantees* as opposed to ensuring correctness in all scenarios. However, such techniques are significantly under-developed when compared to non-probabilistic analyses. The algorithms presented in Chapters 5 and 6 are the first *complete* algorithms for compositional reasoning in order to deal with the state-space explosion problem. However, real software deals with infinite data types (or practically infinite, if physical limitations of the machine are taken into account). It would be interesting to explore SAT/SMT-based methods for probabilistic reasoning. While there exist

techniques for Bounded Model Checking of probabilistic programs (e.g., [26]), we are not aware of *unbounded* verification algorithms. This would require new expressive logics for proofs, appropriate proof-systems, algorithms for constraint solving in presence of probabilities (finding suitable problem formulations as well as solutions), etc.

Bibliography

[1] `https://cpachecker.sosy-lab.org/`.

[2] `http://linuxtesting.org/project/ldv`.

[3] Software Verification Competition. TACAS, 2013. `http://sv-comp.sosy-lab.org/`.

[4] Software Verification Competition. TACAS, 2014. `http://sv-comp.sosy-lab.org/`.

[5] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From Under-Approximations to Over-Approximations and Back. In *TACAS*, 2012.

[6] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig Interpretation. In *SAS*, 2012.

[7] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In *VMCAI*, pages 39–55, 2012.

[8] Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. UFO: Verification with Interpolants and Abstract Interpretation - (Competition Contribution). In *TACAS*, 2013.

[9] Luca de Alfaro, Thomas A. Henzinger, and Ranjit Jhala. Compositional Methods for Probabilistic Systems. In *CONCUR*, volume 2154 of *LNCS*, pages 351–365, London, UK, 2001. Springer-Verlag.

[10] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T.W. Reps, and M. Yannakakis. Analysis of Recursive State Machines. *TOPLAS*, 27(4):786–818, 2005.

[11] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. Analysis of Recursive State Machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[12] Nina Amla and Kenneth L. McMillan. A Hybrid of Counterexample-Based and Proof-Based Abstraction. In *FMCAD*, pages 260–274, 2004.

[13] Nina Amla and Kenneth L. McMillan. Combining Abstraction Refinement and SAT-Based Model Checking. In *TACAS*, 2007.

[14] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.

[15] Dana Angluin and Carl H. Smith. Inductive Inference: Theory and Methods. *ACM Comp. Surv.*, 15(3):237–269, September 1983.

[16] Domagoj Babic and Alan J. Hu. Structural Abstraction of Software Verification Conditions. In *CAV*, 2007.

[17] Christel Baier. On Algorithmic Verification Methods for Probabilistic Systems. Habilitation thesis, Fakultät für Mathematik und Informatik, Universität Mannheim, 1998.

[18] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, MA, USA, 2008.

[19] Thomas Ball and Sriram K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *SPIN*, pages 113–130, 2000.

[20] Gérard Basler, Daniel Kroening, and Georg Weissenbacher. SAT-based Summarization for Boolean Programs. In *SPIN*, pages 131–148, 2007.

[21] Amos Beimel, Francesco Bergadano, Nader H. Bshouty, Eyal Kushilevitz, and Stefano Varricchio. Learning Functions Represented as Multiplicity Automata. *J. ACM*, 47(3):506–530, May 2000.

[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *TACAS*, pages 193–207, 1999.

[23] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On Solving Universally Quantified Horn Clauses. In *SAS*, 2013.

[24] François Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Formal Methods in Programming and their Applications*, pages 128–141, 1993.

[25] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, 2011.

[26] Bettina Braitling, Ralf Wimmer, Bernd Becker, and Erika Ábrahám. Stochastic Bounded Model Checking: Bounded Rewards and Compositionality. In *MBMV*, pages 243–254, 2013.

[27] Rafael C. Carrasco and Jose Oncina. Learning Deterministic Regular Grammars From Stochastic Samples in Polynomial Time. *RAIRO*, 33:1–20, 1999.

[28] Rafael C. Carrasco, Jose Oncina, and Jorge Calera-Rubio. Stochastic Inference

of Regular Tree Languages. *Machine Learning*, 44(1-2):185–197, July 2001.

[29] Rohit Chadha and Mahesh Viswanathan. A Counterexample-Guided Abstraction-Refinement Framework for Markov Decision Processes. *TOCL*, 12(1):1–49, November 2010.

[30] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular Verification of Software Components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[31] Sagar Chaki, Edmund M. Clarke, Nishant Sinha, and Prasanna Thati. Automated Assume-Guarantee Reasoning for Simulation Conformance. In *CAV*, volume 3576 of *LNCS*, pages 534–547, Berlin, Heidelberg, 2005. Springer-Verlag.

[32] Sagar J. Chaki. *A Counterexample Guided Abstraction Refinement Framework for Verifying Concurrent C Programs*. PhD thesis, Pittsburgh, PA, USA, 2005.

[33] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning Minimal Separating DFA's for Compositional Verification. In *TACAS*, volume 5505 of *LNCS*, pages 31–45, Berlin, Heidelberg, 2009. Springer-Verlag.

[34] Alessandro Cimatti and Alberto Griggio. Software Model Checking via IC3. In *CAV*, 2012.

[35] E. Clarke, D. Long, and K. McMillan. Compositional Model Checking. In *LICS*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[36] Edmund M. Clarke. Programming Language Constructs for Which It Is Impossible To Obtain Good Hoare Axiom Systems. *JACM*, 26(1):129–147, 1979.

[37] Edmund M. Clarke. Program Invariants as Fixed Points. *Computing*, 21(4): 273–294, 1979.

[38] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, 2000.

[39] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 2000.

[40] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, 2004.

[41] S. A. Cook. Axiomatic and Interpretative Semantics for an Algol Fragment. In *POPL*, 1977.

[42] D. C. Cooper. Theorem Proving in Arithmetic without Multiplication. *Machine*

Intelligence, pages 91–100, 1972.

[43] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Symbolic Logic*, 22(3):269âĂŞ285, 1957.

[44] Colin de la Higuera and José Oncina. Learning Stochastic Finite Automata. In *ICGI*, volume 3264 of *LNCS*, pages 175–186, Berlin, Heidelberg, 2004. Springer-Verlag.

[45] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[46] Bruno Dutertre and Leonardo de Moura. The Yices SMT Solver. Technical report, SRI International, 2006.

[47] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *SAT*, 2003.

[48] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient Implementation of Property Directed Reachability. In *FMCAD*, pages 125–134, 2011.

[49] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In *CAV*, CAV '00, pages 232–247, 2000.

[50] Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. eVolCheck: Incremental Upgrade Checker for C. In *TACAS*, pages 292–307, 2013.

[51] Lu Feng, Tingting Han, Marta Kwiatkowska, and David Parker. Learning-based Compositional Verification for Synchronous Probabilistic Systems. In *ATVA*, volume 6996 of *LNCS*, pages 511–521, Berlin, Heidelberg, 2011. Springer-Verlag.

[52] Lu Feng, Marta Kwiatkowska, and David Parker. Automated Learning of Probabilistic Assumptions for Compositional Reasoning. In *FASE*, volume 6603 of *LNCS*, pages 2–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[53] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The Spirit of Ghost Code. In *CAV*, pages 1–16, 2014.

[54] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME*, pages 500–517, 2001.

[55] M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based Unbounded Symbolic Model Checking Using Circuit Cofactoring. In *ICCAD*, pages 510–517, 2004.

[56] Sicun Gao. *Computable Analysis, Decision Procedures, and Hybrid Automata: A New Framework for the Formal Verification of Cyber-Physical Systems*. PhD

thesis, Carnegie Mellon University, 2012.

[57] Sicun Gao, Soonho Kong, Wei Chen, and Edmund M. Clarke. Delta-Complete Analysis for Bounded Reachability of Hybrid Systems. *CoRR*, abs/1404.7171, 2014.

[58] Pedro Garcia and Jose Oncina. Inference of Recognizable Tree Sets. Research Report DSIC - II/47/93, Universidad Politecnica de, 1993.

[59] Mihaela Gheorghiu Bobaru, Corina S. Păsăreanu, and Dimitra Giannakopoulou. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In *CAV*, volume 5123 of *LNCS*, pages 135–148, Berlin, Heidelberg, 2008. Springer-Verlag.

[60] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *POPL*, pages 43–56, 2010.

[61] E. Mark Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.

[62] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, 1997.

[63] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing Software Verifiers from Proof Rules. In *PLDI*, pages 405–416, 2012.

[64] Alberto Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, January 2012.

[65] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically Refining Abstract Interpretations. In *TACAS*, 2008.

[66] Aarti Gupta, Malay K. Ganai, Zijiang Yang, and Pranav Ashar. Iterative Abstraction using SAT-based BMC with Proof Analysis. In *ICCAD*, pages 416–423, 2003.

[67] Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. Automated Assumption Generation for Compositional Verification. *FMSD*, 32(3):285–301, June 2008.

[68] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested Interpolants. *SIGPLAN Not.*, 45(1)(1):471–482, 2010.

[69] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and An-

dreas Podelski. Ultimate Automizer with SMTInterpol - (Competition Contribution). In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 641–643. Springer, 2013. ISBN 978-3-642-36741-0.

[70] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. *SIGPLAN Not.*, 37(1):58–70, 2002.

[71] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. of POPL*, pages 58–70, 2002.

[72] Holger Hermanns, Björn Wachter, and Lijun Zhang. Probabilistic CEGAR. In *CAV*, volume 5123 of *LNCS*, pages 162–175, Berlin, Heidelberg, 2008. Springer-Verlag.

[73] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969.

[74] Kryštof Hoder and Nikolaj Bjørner. Generalized Property Directed Reachability. In *SAT*, 2012.

[75] Alexander Ivrii, Arie Matsliah, Hari Mony, and Jason Baumgartner. IC3-Guided Abstraction. In *FMCAD*, 2012.

[76] Himanshu Jain, Franjo Ivančić, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. Using Statically Computed Invariants Inside the Predicate Abstraction and Refinement Loop. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 137–151, 2006.

[77] Ranjit Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *TACAS*, 2006.

[78] Anvesh Komuravelli, Corina S. Păsăreanu, and Edmund M. Clarke. Assume-Guarantee Abstraction Refinement for Probabilistic Systems. In *CAV*, 2012.

[79] Anvesh Komuravelli, Corina S. Păsăreanu, and Edmund M. Clarke. Learning Probabilistic Systems from Tree Samples. In *LICS*, 2012.

[80] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. Automatic Abstraction in SMT-based Unbounded Software Model Checking. In *CAV*, 2013.

[81] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, 2014.

[82] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In G. Gopalakrishnan and S. Qadeer, editors,

CAV, volume 6806 of *LNCS*, pages 585–591, Berlin, Heidelberg, 2011. Springer-Verlag.

[83] Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Assume-Guarantee Verification for Probabilistic Systems. In *TACAS*, volume 6015 of *LNCS*, pages 23–37, Berlin, Heidelberg, 2010. Springer-Verlag.

[84] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 427–443. Springer, 2012. ISBN 978-3-642-31423-0.

[85] Bing Li and Fabio Somenzi. Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking. In *TACAS*, 2006.

[86] R. Loos and V. Weispfenning. Applying Linear Quantifier Elimination. *Computing*, (36(5)):450–462, 1993.

[87] Hua Mao et al. Learning Probabilistic Automata for Model Checking. In *QEST*, pages 111–120, Washington, DC, USA, 2011. IEEE Computer Society.

[88] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, 2003.

[89] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, 2006.

[90] Kenneth L. McMillan. Lazy Annotation for Program Testing and Verification. In *CAV*, 2010.

[91] Kenneth L. McMillan and Nina Amla. Automatic Abstraction without Counterexamples. In *TACAS*, 2003.

[92] Kenneth L. McMillan and Andrey Rybalchenko. Solving Constrained Horn Clauses using Interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.

[93] Robin Milner. An Algebraic Definition of Simulation between Programs. Technical report, Stanford, CA, USA, 1971.

[94] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[95] Tobias Nipkow. Linear Quantifier Elimination. *J. Autom. Reason.*, 45(2): 189–212, 2010.

[96] Arlindo L. Oliveira and Joao P. Marques-Silva. Efficient Search Techniques for the Inference of Minimum Size Finite Automata. In *SPIRE*, pages 81–89.

IEEE Computer Society Press, 1998.

[97] Jose Oncina and Pedro Garcia. Identifying Regular Languages In Polynomial Time. In *ASSPR*, volume 5, pages 99–108. World Scientific, 1992.

[98] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. In *LMCS*, volume 13 of *NATO ASI*, pages 123–144. Springer-Verlag, 1985.

[99] Corina S. Păsăreanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to Divide and Conquer: Applying the L* Algorithm to Automate Assume-Guarantee Reasoning. *FMSD*, 32(3):175–205, June 2008.

[100] Michael O. Rabin. Probabilistic Automata. *Information and Control*, 6(3): 230–245, 1963.

[101] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, pages 49–61, 1995.

[102] Roberto Segala and Nancy Lynch. Probabilistic Simulations for Probabilistic Processes. *Nordic J. of Computing*, 2(2):250–273, June 1995.

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

[104] Wen-Guey Tzeng. Learning Probabilistic Automata and Markov Chains via Queries. *Machine Learning*, 8(2):151–166, March 1992.

[105] Yakir Vizel, Orna Grumberg, and Sharon Shoham. Lazy Abstraction and SAT-based Reachability for Hardware Model Checking. In *FMCAD*, 2012.

[106] Lijun Zhang. *Decision Algorithms for Probabilistic Simulations*. PhD thesis, Universität des Saarlandes, 2008.